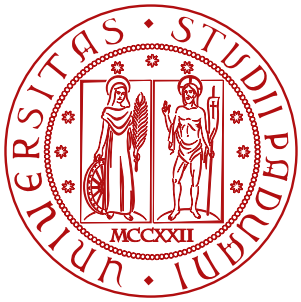

University of Padova

Department of Information Engineering

Master's degree in Computer Engineering

OPTIMIZATION OF WAREHOUSE INVENTORY PLACEMENT VIA PICK-TIME PREDICTION

Supervisors

Prof. Andrea Alberto Pietracaprina

Prof. Geppino Pucci

Prof. Matteo Ceccarello

Company tutors

Dott. Francesco Locascio

Ing. Carlo Vecchio

Candidate

Mattia Borgo

2050778

March 27th, 2025

Academic year 2024–2025

ABSTRACT

The aim of this thesis is the analysis of the problem and development of an optimization system for warehouse inventory positioning in a commercial setting in which such positioning is not only bound by limits on pick time but also on the order in which items are ultimately stored in picking carts. To that end a recurrent neural network was employed to predict picking time for a given positioning by learning statistical distributions for such times from timing data related to past orders from the same warehouse. This last system shows an average error between ground truth and output distribution average of about 22% on average, whereas the new positioning of items in the new warehouse allows for picking times to match those of the old warehouse while significantly increasing adhererence to the ordering constraints.

SOMMARIO

Scopo di questa tesi è l'analisi del problema e lo sviluppo di un sistema di ottimizzazione del posizionamento di articoli in un magazzino commerciale soggetto a vincoli di tempo di picking oltre che di ordinamento di articoli in carrello. A tale fine è stata applicata una rete neurale ricorrente per la predizione dei tempi di picking, che ha appreso una distribuzione statistica degli stessi da rilevazioni su ordini precedenti relativi allo stesso magazzino. Quest'ultima riporta uno scarto medio tra valore vero e media della distribuzione del 22% in media, mentre la nuova disposizione degli articoli nel nuovo magazzino permette di ottenere tempi di picking uguali a quelli del vecchio magazzino a fronte di un accordo molto migliore ai vincoli di ordinamento.

CONTENTS

1 Introduction 1
1.1 Organization of the work . . . 2
2 Setting 5
2.1 Warehouse structure . . . 5
2.1.1 Shelves . . . 6
2.1.2 Old warehouse structure . . . 8
2.1.3 New warehouse structure . . . 9
2.2 Picker behaviour . . . 9
2.2.1 Old warehouse zoning . . . 10
2.2.2 New warehouse path . . . 12
2.3 Picking constraints . . . 12
2.4 Prediction and optimization . . . 13
3 Data pre-processing 15
3.1 Input format . . . 15
3.1.1 Order reports . . . 16
3.1.2 Item database . . . 17
3.1.3 Position database . . . 18
3.2 Data cleaning . . . 19
3.2.1 Duplicate act aggregation . . . 20
3.2.2 Start and end time fixing . . . 20
3.3 Anonymization and warehouse independence . . . 21
3.3.1 Picker feature extraction . . . 21
3.3.2 Position data . . . 23
3.4 Numerical representation . . . 23
3.4.1 Time encoding . . . 24
4 Prediction model 29
4.1 Problem statement . . . 29
4.2 Network structure . . . 31
4.2.1 Deep neural network . . . 32
4.2.2 Recurrent neural networks . . . 33
4.2.3 Gated recurrent units . . . 35
4.2.4 Overall structure . . . 36
4.3 Training . . . 39
4.4 Results . . . 41
5 Optimization model 47
5.1 Problem statement . . . 48
5.2 Metaheuristics . . . 49
5.2.1 Simulated annealing . . . 50
5.2.2 Genetic algorithm . . . 51
5.3 Evaluation function . . . 53
5.4 Evolution . . . 55

5.4.1 Representation 55
5.4.2 Crossover 56
5.4.3 Mutation 58
5.4.4 Selection 59
5.5 Results 61
5.5.1 Simulated annealing 63
5.5.2 Genetic algorithm 65
6 Conclusions 71
6.1 Future research 71

Bibliography 73

LIST OF FIGURES

Figure 2.1 Ideal warehouse structure with labeled components. 5
Figure 2.2 Aisle view of the kinds of shelves under consideration. 6
Figure 2.3 Cross-section of a roller shelf. 7
Figure 2.4 Old warehouse structure mock-up. 8
Figure 2.5 Sample picking paths for Figure 2.4. 11
Figure 3.1 Order-to-order time difference histogram. . . . 22
Figure 3.2 Gaussian radial basis functions for $n = 6$. . . . 26
Figure 3.3 Sum of adjacent Gaussian radial basis functions. 26
Figure 4.1 Structure of a neural network layer. 32
Figure 4.2 Schematic structure of a recurrent neural network (RNN). 33
Figure 4.3 Internal structure of a gated recurrent unit (GRU) cell. 36
Figure 4.4 Structure of the prediction network. 37
Figure 4.5 Order time histogram for acts between adjacent positions. 38
Figure 4.6 P-P plot for acts of Figure 4.5. 38
Figure 4.7 Loss and validation loss for 32 units with $\eta = 10^{-4}$. 42
Figure 4.8 Predictions for intra-area travel distances. . . . 44
Figure 4.9 Predictions for inter-area travel distances. . . . 44
Figure 5.1 Example of representation for an assignment c. 55
Figure 5.2 Cycle crossover example. 57
Figure 5.3 Chain crossover example. 58
Figure 5.4 Stochastic universal sampling example. 60
Figure 5.5 Trajectories as a function of $\frac{\lambda}{\mu}$. 67
Figure 5.6 Two-dimensional histogram for $\nu = 100$. 69
Figure 5.7 Two-dimensional histogram for $\nu = 1000$. . . . 69

LIST OF TABLES

Table 3.1 Example of raw order report data. 15
Table 3.2 Example of item and position database data. . 17
Table 4.1 Order and act counts for the dataset. 41

Table 4.2 Validation loss matrix for different hyperparameters. 43
Table 5.1 New warehouse item database statistics. 61
Table 5.2 New warehouse roller constraint statistics. . . 61
Table 5.3 Baseline statistics for the optimization problem. 63
Table 5.4 simulated annealing (SA) results for $\nu = 100$. . 64
Table 5.5 SA results for $\nu = 1000$. 64
Table 5.6 genetic algorithm (GA) results for $\nu = 100$. . . . 66
Table 5.7 GA results for $\nu = 1000$. 66

LIST OF ALGORITHMS

Algorithm 4.1 The RMSprop algorithm in the context of our network. 40
Algorithm 5.1 The threshold accepting SA variant. 51
Algorithm 5.2 Generic GA. 52

ACRONYMS

ANN	artificial neural network
BPTT	back-propagation through time
GA	genetic algorithm
GRU	gated recurrent unit
LSTM	long short-term memory
ML	machine learning
RFID	radio-frequency identification
RNN	recurrent neural network
SA	simulated annealing
SKU	stock-keeping unit
WMS	warehouse management system

1 INTRODUCTION

Order picking is the cornerstone of warehouses. It is estimated that 65% of warehouse operating costs is due to handling incoming orders, assigning them to operators, and routing them through the warehouse to pick items (Coyle, Bardi and Langley, 1996); half of the time spent processing orders is in fact the operator's travel time (Tompkins et al., 2010). It may therefore be said that optimizing this process can lead to major improvements in both warehouse productivity and spending.

Much research has been conducted on this subject, and as stated in Roodbergen and de Koster, 2001, four optimization avenues have been identified: routing, warehouse zoning, order batching, and storage assignment. Storage assignment is the most impactful on the whole process among those four, as the other three are directly dependent on it for their functioning (and thus optimization) and so its modification has long-reaching consequences.

Despite its importance, traditional methods for storage assignment do not directly strive to minimize costs or travel time, though it is often their main concern; they are instead based on engineering metrics that act as a stand-in for them but are directly derived from either experience or past order data. In particular, following the classification in Riccardo Accorsi and Bortolini, 2012, we note two main strategies of this kind:

- classification of items according to their turn-over rate and assignment to areas of the warehouse that are seen first in the picking process, and similarly for other indices that relate to an item's "popularity."

- pattern mining, and more specifically looking for items that are often ordered together to find clusters that should be placed close to each other or by dividing items into *classes*, each of which is associated with a separate area in the warehouse (Ansari and Smith, 2020; Zhang, 2016); this may also integrate other logistically relevant metrics such as turn-over (Mirzaei, Zaerpour and de Koster, 2021).

Furthermore, the problem is often analyzed in its *slotting* form, that is, when deciding where a new item should be placed in a pre-existing warehouse, as opposed to its full reorganization.

More recent research, as reviewed in Islam and Uddin, 2023 only partly focuses on the latter strategy. Most of it is instead dedicated to modeling warehouse operations then directly minimizing travel

time or distance in this form. This approach is able to include other features of items that are relevant to picking, such as weight; as stated in van Gils et al., 2019, this is very desirable, as it provides noticeably superior assignments.

At the same time, models derived in this way are almost invariably *linear* and require significant work in devising features to be used as coefficients. As shown in Gino Marchet and Perotti, 2015, however, a majority of warehouses still employs manual pickers, whose behavior is more varied than such a model would allow; this limitation was noted by Kabak and Güleç, 2022, who then propose an artificial neural network (ANN)-based predictor for picking times.

Our chosen approach is thus to predict picking times by modeling picker behavior by learning from past data with the aid of a recurrent neural network based on GRUs (Cho et al., 2014), enabling us to evaluate how different item placements in the warehouse affect such times. This predictor is then used by a metaheuristic-based optimization model to approximate optimal item positioning with respect to picking time for known orders as well as fitting within logistical constraints on picking order.

This approach was developed for and applied to a real warehouse in Northeast Italy; their interest in the subject was spurred by the need to reorganize their stock after renovations. A significant part of our efforts relates to handling of data generated by the warehouse so that it may be used by the network, as well as the encoding of the specific logistical constraints and their effect on generated item arrangements.

Our work is further complicated by the fact that the data at our disposal was relative not to the same warehouse for which optimization was to be computed, but rather to an earlier status of the warehouse prior to renovations; as a result care had to be taken not to base prediction on any feature inherent to the structure of either. This is not generally an area that has received much if any interest in research, as it only occurs in extremely specific circumstances; it nevertheless poses a significant challenge for prediction and shaped much of the structure of our approach.

Ultimately, this work endeavors to analyze the specific case that spurred this research while providing a more general outlook on the problem to be solved and on the framework used.

1.1 ORGANIZATION OF THE WORK

Chapter 2 provides basic definitions and information about the handling of warehouses, as well as further details regarding the old and new warehouse and their differences. In the chapter we describe how warehouses are structured from a design and logistics point of view, as well as how wares, or stock-keeping units (SKUs), can be arranged

in them, and how operators — *pickers* — travel through them to pick items and fulfill orders. Finally, we introduce the concrete problem, and justify our split prediction–optimization approach.

In Chapter 3 we describe how the raw data that describes the day-to-day operations of the warehouse is processed for use in a machine learning (ML) setting for both the prediction and the optimization steps. This includes removing inconsistencies, fixing issues caused by incorrect or incomplete tracking of picker behavior, as well as a crucial anonymization and warehouse-independence step that allows us to use the predictor model trained on old warehouse data for its new layout. The output of this step is a dataset in purely numerical form.

Chapter 4 introduces the *picking time prediction problem* in both its order-level and its act-level form. The chapter continues by building up to the description of the model, from ANNs to RNNs and finally to GRU and the structure of the network. Finally, the training process is described and results are presented both for training and for inference on a test set.

In Chapter 5 we briefly mention past research on the storage assignment problem and describe it as a full-warehouse problem as opposed to its slotting form. We then introduce our two chosen metaheuristics for the problem — SA and GA — after a description of how the operations that they expect — mutation and cross-over — can be implemented for storage configurations, and of how the predictor model is used to evaluate each such configuration while including constraints. Results are presented for overall picking time as well as adherence to constraints.

Lastly, Chapter 6 summarizes the results of this work, highlights the areas in which improvements could be made — whether the input data, the predictor or the optimizer — and suggests future avenues of inquiry.

2 SETTING

In this chapter we describe all elements that relate to the structure, organization and handling of a warehouse, as well as several key details of the problem we are tackling.

This work deals with one warehouse that has gone through a renovation process that changed much of its structure. Data for its operation was collected only for the *old* warehouse, as the *new* warehouse was under construction at the time. As a result of this, we are forced to split this description in two parts for all but the last section, according to whether we are describing the old or the new warehouse, as the two differ in many significant respects.

2.1 WAREHOUSE STRUCTURE

It is sufficient for our purposes to describe what is arguably the "core" of the warehouse, i. e. the area where goods are stored, and therefore also the area in which picking occurs.

The ideal warehouse structure is shown in Figure 2.1. The smallest repeated units are *shelves*, which face *aisles* through which warehouse operators move, and *bays* that allow each shelf to have a unique coordinate triplet: aisle–bay–side. It is also common for some bays to be missing from the floorplan or to be completely hollow on the bottom levels to make space for passageways and underpassages that run perpendicular to aisles; this can happen for safety reasons as well as to provide greater maneuverability for operators that make use of forklifts and other motorized equipment.

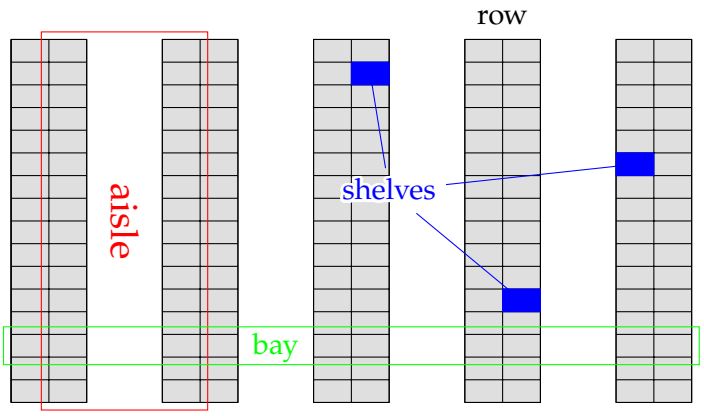

Figure 2.1: Ideal warehouse structure with labeled components.

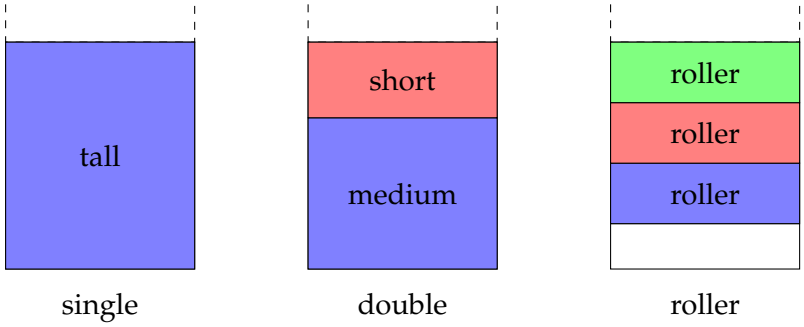

Figure 2.2: Aisle view of the kinds of shelves under consideration.

Of lesser importance for logistics but still useful for descriptive purposes is the *row*: a single structure composed of multiple shelves mounted and packed together so as to face either one or two aisles — in the labeled example, counting left to right, the fourth row separates the third and fourth aisles, and its shelves face one or the other. Rows are almost invariably made of shelves that face one and only one aisle each, as in Figure 2.1; this is to make each shelf accessible from only one physical position and so prevent multiple operators from attempting to manipulate items at the same time (whether at the upper or ground levels).

2.1.1 Shelves

Shelves are so named because they extend vertically and provide shelving space for stocked items at higher levels, whereas rungs reachable from ground level are where items get picked up; the structure of each shelf may differ there as a function of what items are to be picked from it. Figure 2.2 shows a schematic structure of a few kinds of shelves, namely the ones that are found in both warehouses to be described. They are depicted from the point of view of an operator, i. e. as seen from the aisle; dashed lines indicate that the shelves continue upwards. As in the figure, the region where picking takes place has the same overall height for all kinds of shelves — usually around two metres from the floor.

In the following, we shall make a distinction between SKUs and items. An SKU is a class of items which, for the purposes of the warehouse, are considered the same; each of them is replaceable with one another and they are all stocked in the same place. As a result each item "belongs" to a single SKU.

Single shelves stock only one SKU; thus there is also only one slot for items at ground floor, called a *tall module*, as all vertical space is dedicated to the single SKU. Items stocked in these shelves may either be very tall and fit nowhere else or be ordered so often that it is

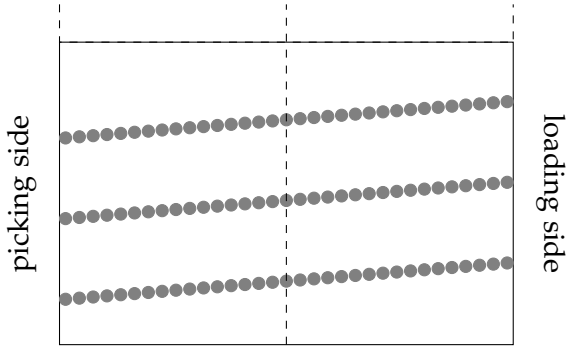

Figure 2.3: Cross-section of a roller shelf.

necessary for overall warehouse performance to have a great amount of items ready for picking.

Double shelves stock two SKUs and may provide a different amount of upper shelf space between the two of them, as required by order volume; the two slots that result are called *medium modules* and *short modules* as the vertical space they occupy is different. To minimize weight both for the shelf and for operators that have to pick packages up manually, medium modules — the taller of the two — are found at ground level, and short positions occupy the first rung, as in the figure.

The structure of *roller* shelves is more involved. As Figure 2.2 shows, from a schematic point of view they have a certain number of vertically spaced but otherwise indistinguishable modules, also called *roller* modules; they, however, feature a finer structure than may be seen from such a view.

In the case of two-sided rows each roller module at ground level spans *two* shelves placed side-by-side: items are picked from one aisle and are loaded onto rollers — that is, reinforced cylinders mounted parallel to aisles so as to be free to rotate on their axis — from the other, at which point they will roll to the other side by way of gravity. A cross-section of such an arrangement is shown in Figure 2.3, where rollers are displayed as gray circles.

As a result of this arrangement, only one side (aisle) may be picked from, whereas the other is used exclusively for loading. The key point, however, is that rollers require access on both sides; they can still be part of single-sided rows, provided that there is a way to load items from the back — an area where no other shelf is otherwise accessed.

Finally, each of these shelf types may be further split horizontally at ground level to provide space for more kinds of SKUs, at the expense of reduced area dedicated to each of them — both for the picking area and for stock in upper level. For both warehouses, single and double shelves are split into at most 3 separate areas, where the exact amount depends on the structure of the building — parts of some shelves are missing as they intersect load-bearing columns for the ceiling; single

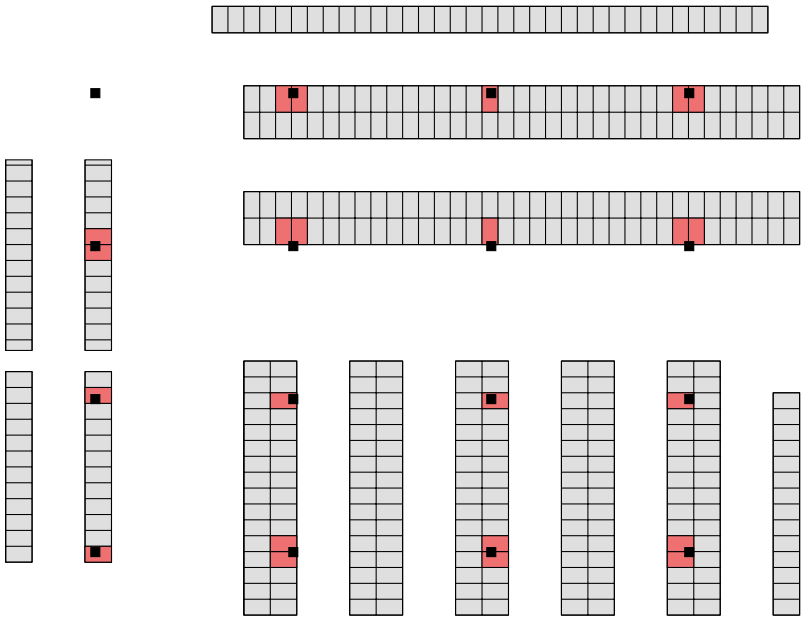

Figure 2.4: Old warehouse structure mock-up.

shelves thus offer at most 3 picking positions, and double shelves at most 6. Roller shelves are always split into 6 separate horizontal areas for a total of 18 picking positions.

2.1.2 Old warehouse structure

The old warehouse was split into multiple areas each of which was structured approximately as in Figure 2.1, though not all aisle–bay combinations were present for each area: some bays would begin to have shelves well after the first aisle and could end well before the last, and as mentioned earlier regarding missing parts of some shelves, columns and other unchangeable architectural features of the warehouse building may interfere with the ideal regular arrangement of rows and shelves.

This more complex arrangement of shelves is due to a few reasons. The main one is that the haphazard structure comes from multiple expansions of the warehouse: as demand grew new areas would be added to store additional SKUs, but as this happened in stages, no greater planning could take place, and the overall structure was left segmented. Another less pressing issue is that the warehouse in question, both old and new, stores *perishable items*, and in particular meat and dairy products. Some such items may therefore involve different handling of their packages or may have different storage requirements, in which case a separate area would be dedicated to different classes of SKUs.

Figure 2.4 shows a warehouse of a structure conceptually similar to the one under consideration. The area on the left is composed of a

single aisle split in the middle by a passageway; both of its rows are single-sided. The area on top has its aisles running perpendicular to the other two and its topmost row is also single-sided — this happens regularly for rows that are mounted near or attached to a wall of the building. Lastly, both areas on the right show how some shelves may be missing with respect to the ideal structure, and when they do it is most commonly at the beginning or at the end of each aisle. In particular, shelves displayed in red would overlap with a column, shown as a black square; it is these shelves that are either not present at all, or if they are, may have been halved to fit in the reduced floorspace.

2.1.3 New warehouse structure

The new warehouse overhauled a great part of the structure of the old one, making it far more uniform and much closer to the one in Figure 2.1. This was done by extending one of the areas to fill the entire warehouse — an area of the same structure as the one on the bottom right of Figure 2.4. While still missing some aisle–bay combinations — indeed, some bays only consist of a few aisles at the extreme ends — the final structure is almost exactly the one we described as "ideal".

As mentioned earlier, the issue of peculiar handling and storing of different kinds of perishables was less of a concern, and so "merging" all areas into one (from a logistics point of view) posed no particular difficulty or additional constraint on the structure of the new warehouse.

2.2 PICKER BEHAVIOUR

A *picker* is an operator whose job is to traverse the warehouse and fulfill an order by picking items from their pre-determined locations in the warehouse. There are many choices inherent in the expected behavior of a picker, which are mostly summarized by the route they take to pick each item or all items at once.

Warehouses typically employ a warehouse management system (WMS); this piece of software handles incoming orders as well as resupplying and other upkeep automatically. For the purposes of picking it is sufficient to describe the WMS's handling of orders, and as stated above, the way that it translates a given order into a route for the picker; this is part of its *picking strategy*, which also includes how the WMS splits, batches and times orders, if it has such capabilities. As we do not deal with "surface" orders — those coming from the stores themselves — but rather from the actual orders as seen by pickers we may safely ignore the latter portion of the strategy.

Pickers handle orders in *picker-to-parts* warehouses, as opposed to *automated* warehouses, which instead employ robots for the same purpose, and *parts-to-picker* warehouses, where conveyor belts or other transportation is used to bring goods to pickers, who then assemble the order. It is for this reason that the shelf types described earlier are organized differently at ground floor: while a picking robot may move vertically and pick at any height, human pickers have limited reach and will only be able to pick items if they are available at ground floor.

For this reason, the WMS is limited in its decisions to a relatively simple and uniform route for pickers, for reasons of safety as well as to limit confusion and traffic in the warehouse. As an example, pickers cannot always be allowed to take the shortest route to the next picking position in their order, as that might cross a passageway for restocking forklifts to pass through, or it may cause them to collide with another picker going the opposite way.

Thus the route generated by the WMS for any order is almost the same: pickers cross the warehouse through a fixed, predetermined path moving in a single direction; they pick items in the order they see them, but they are not generally allowed to turn back. Special circumstances may force them to tread the same path backwards, most commonly when an item is out of stock at picking (i. e. ground) level and has not yet been restocked from the upper shelves once they come across it — though such cases are usually predicted and managed ahead of time by the WMS.

Pickers make use of a *roll container* (hereafter a *roller*) that they move through the warehouse and into which they load items for a single order. This roller may be further processed after picking is complete, but it is ultimately the result that is shipped to the destination, i. e. the point of sale.

It is also the WMS's job to *monitor* each picker's behavior and check that each item has been loaded in the expected order. To do so a hand-held computer is provided to each picker that communicates directly with the management system; through this device they are notified of the next item to pick and with it they scan items as they pick them, be it a barcode or tags that support radio-frequency identification (RFID). While the ideal case would be for scanning to take place immediately after the item is loaded onto the roller, this does not always happen: as functioning of the WMS itself is not hindered by this delay, there is no reason for pickers or the system itself to ensure this property; we will see later in Chapter 3 that this is a non-trivial issue for our purposes.

2.2.1 Old warehouse zoning

The old warehouse was split into three disjoint *zones*; each order would have its SKUs grouped by zone, splitting it into multiple *sub-orders* that would then be handled by one picker per zone. Each picker followed

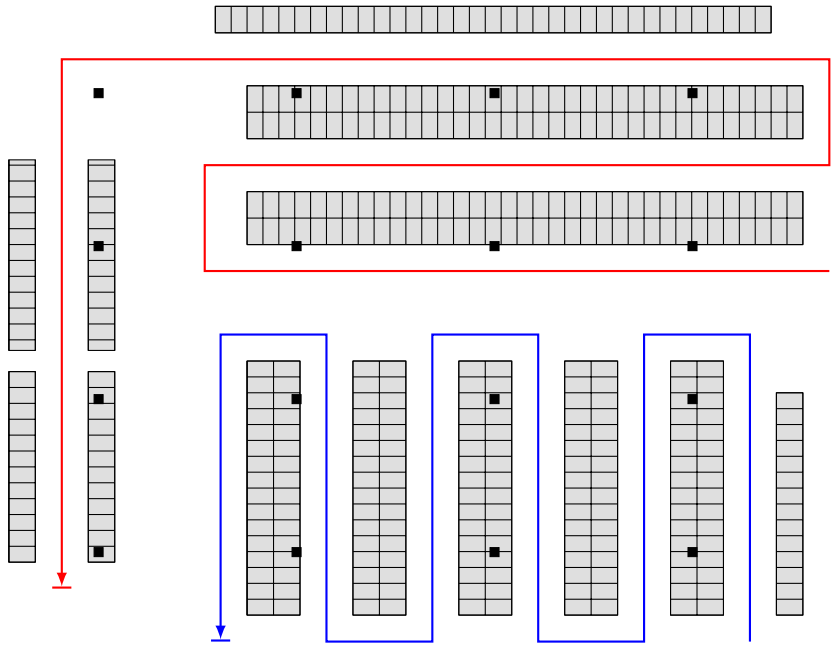

Figure 2.5: Sample picking paths for Figure 2.4.

a set path, as described above, and would not cross zone boundaries to pick other items for the same order. Once this was done, sub-orders would be merged into one roller. Zones comprised one or more of the areas described in Section 2.1.2, but no area was be split between multiple zones.

There were two main reasons for this division. On one hand, routing pickers through every single area of the old warehouse would have led to a path of unreasonable length, as the area-based structure of the old warehouse would have required multiple parts of this same path winding around each other — and it must be kept in mind that this path would also have to abide by the constraint of only passing through each aisle or corridor in one and only one direction. On the other, as mentioned in Section 2.1.2, different areas held different kinds of SKUs, thus some sub-orders may have ended up empty if no items from that zone were ordered at all; this could speed up the picking process.

Figure 2.5 shows a similar arrangement for the warehouse in Figure 2.4. The red and blue paths are followed by pickers of two separate zones. The main pattern for each area is that of a "snaking" route that traverses each aisle in one specific direction, where the last aisle of one area is connected to the first of the next area by a *corridor* which extends outside of both — in the figure, this happens on the top left corner for the red path, whereas the blue path requires no corridor as it only spans a single area.

It is of course possible, although largely irrelevant for our purposes, to assign a single picker to multiple sub-orders and let them handle

each of them serially, i. e. one at a time, moving from one zone to the next.

2.2.2 New warehouse path

The new warehouse, as described in Section 2.1.3, is comprised of a single area; as a result every picker follows *the same* path, whose shape is akin to the blue path in Figure 2.5, and there is no splitting of orders into sub-orders.

While this undoubtedly makes warehouse management and picking easier, we shall see in Chapter 5 that it also poses a significant issue for optimization: sub-orders and therefore picking times cannot be directly compared between old and new warehouse, as an order for the new warehouse will be handled by a single picker in one pass, whereas the same order for the old warehouse would be split into sub-orders of wildly differing sizes according to the positioning of SKUs.

2.3 PICKING CONSTRAINTS

While the WMS, as employed, sorts items inside the given order in a strict manner as described above, it is also true that items may not be placed arbitrarily inside the warehouse; rather, pickers and managers have to consider further *constraints* when it comes to placing and picking items. There is a great deal of choices when it comes to which constraints to consider in this scenario; we will describe those that we have been exposed to in the course of this research.

The most obvious constraint is the one that relates to the *module* that a given item can fit in. Since items may have different sizes, not only would it be impossible to fit a large one inside a shorter module, but it would also be wasteful to place small items inside large modules, especially if the latter are needed only in limited quantities.

Another constraint which is purely of a logistical nature has to do with item *grouping* as it relates to the point of sale: for example, some items may be on display on store shelves, whereas others might need a refrigerated counter. While this is a distinction that is immaterial to the warehouse itself, stores prefer that items that share the same group be close together inside the final roller, so as to minimize the time required by the clerks to restock shelves and counters — if possible, in a single tour of the point of sale — without moving from one to the other multiple times.

The remaining two constraints that we have examined are best described together; they are both physical limitations to the arrangement of items inside the roller. One of them is the requirement that items that are heavier be placed at the bottom of the roller, i. e. at the beginning of the picking sequence for the order; in other words, *weight* is a

constraint. Another is that items that are considered more *fragile* be placed at the top of the roller, i. e. as late as possible in the picking sequence. In both cases, failure to abide by this rule may lead to items breaking or collapsing on each other.

Not all of the constraints above are of equal importance. Modules are strictly required to store specific items, and as such that constraint cannot be loosened in any significant way. On the other end of the spectrum, grouping items together is merely a desirable property, and it may be followed more or less as needed. Finally, the two physical constraints may be loosened in the sense that there exist ways to prevent collapse of the roller (e. g. movable support structures that can be attached to divide the roller in two vertical sections); however, these are expensive both economically and in terms of efficiency, the latter due to the time required to set them up and train pickers in their use. It is also true that not all such rules can always be followed at the same time, as they can and will conflict with one another, and we shall see how this occurs for the warehouse in question in Chapter 5.

What is most desirable, then, is a compromise: a placement of items that broadly follows these constraints, breaking them as necessary for a given goal.

2.4 PREDICTION AND OPTIMIZATION

As mentioned at the beginning of this chapter, the warehouse in question underwent renovations; the original placement of SKUs could not be applied directly to the new structure of the warehouse, so a new placement had to be devised. It was already stated in Section 2.2.1 that zoning potentially improves picking speed due to splitting orders, but as the new warehouse only consisted of a single zone this was no more; thus the main goal for the new item placement was minimizing picking time, and in particular doing so by making use of the large amount of information stored by the WMS about past orders and how pickers dealt with them. However, as described in Chapter 1, even when past research attempted to minimize this metric directly, it favored linear models for its estimation that rarely made use of data related to picker movements

Modeling human behavior is far more complicated than a linear model would allow, however; this is mainly due to the great *unpredictability* in their behavior. Pickers may slow down or speed up the picking process for a multitude of reasons such as tiredness or the total weight of the roller they are servicing; they may deviate from the exact path given to them to short-cut the distance to the next item, sometimes by noticeable amounts — though still abiding by warehouse rules regarding directionality of travel; they may even take

breaks while an order is still pending; and as mentioned in Section 2.2, they may be late in notifying the WMS that an item has been picked.

Our chosen approach was then to split the problem of *prediction* from that of optimization. A prediction model is used to predict the time that a picker spends servicing a certain order, i. e. its *picking time*; this predictor can then be used by the optimization model to obtain an estimate of how long it would take for a picker to service any given order for any candidate SKUs placement. This allows the optimization model to find a placement that that minimizes overall (predicted) picking time over a representative set of orders while also striving to adhere to the constraints described in Section 2.3. This approach was proposed, though with different methods for solving either part of it, by Daria Battini and Sgarbossa, 2015; they also lamented the inability of most studies to either generalize to realistic scenarios or apply to any scenario at all, concerns that would be mitigated by use of general data-driven methods for pick time prediction.

The benefits of such an approach are manifold. First of all, as the prediction model is extremely general, it is possible to make use of a ML model trained on past picker data to tune predictions to how pickers behave in the specific warehouse. Secondly, the prediction model has additional uses beyond the scope of optimizing SKU placement: one could analyse differences in picking the same order at different times of the day, or varying any other feature in input to the predictor, to optimize working conditions for pickers. Finally, use of the same kind of optimization model as the ones in the literature allows us to include other goals, such as the picking constraints above, without issue.

3 DATA PRE-PROCESSING

In this chapter we describe the format of the data as exported from the WMS and the process through which it is encoded for use in a ML process.

The data refers specifically to the old warehouse, but is to be used to optimize the new one; thus, additional care must be taken to make it *independent* from its exact structure and contents. Furthermore, in order to be able to reuse a model trained on the output data even in the case of new pickers being added to or leaving the workforce, an *anonymization* step is performed, which extracts information that would require knowledge of the identity of each picker before discarding it.

The description given below is split into logical processing steps; their implementation does not necessarily follow the same order; the latter is geared towards discarding as much information as early as possible so as to work with as little data as possible on each step. We have chosen to include below a few hints on the earliest points at which some of the final columns can be computed.

3.1 INPUT FORMAT

The data we were provided with was split into three separate tables; their format is described below.

ORD	PCS	WT	SKU	PICKER	P/P	LOCATION	DIST	START	END
1	16	0.8	33	USER1	4	03-02-L-1	12.2	13:12:05	13:12:10
1	1	0.1	4341	USER1	1	03-05-R-3	12.6	13:12:10	13:12:13
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
196	5	2.5	76	USER2	1	09-33-L-2	60.9	21:45:03	21:45:33
196	2	0.4	691	USER2	2	10-20-L-1	70.3	21:46:50	21:46:51
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 3.1: Example of raw order report data.

3.1.1 Order reports

The raw data as generated by the WMS takes multiple forms. The bulk of it comes from weekly *order reports*; our dataset included 24 weeks of reports, for a total of over 7,000,000 rows of data.

An example of the structure of a report is given in Table 3.1. Each row of the input table describes a single *picking act*, in which a picker moves around the path given to them by the WMS towards the place where some SKU is stored and picks as many items as required. Data for a full sub-order is therefore composed of multiple rows, one (or, as we will see, possibly more) per SKU. The meaning of the columns is as follows:

- `ORD` is the order number, which allows one to recognize which acts belong to the same sub-order.
- `PCS` is the total number of pieces, i. e. items, picked in the act.
- `WT` is the total weight, in kilograms, of all items picked. This value may be zero for items whose weight was never input into the WMS.
- `SKU` is a numerical code that identifies which SKU was picked in the act.
- `PICKER` is an opaque identifier for the picker who communicated with the WMS regarding the act. It is not technically necessary for this value to be the same for all acts for the same order, and indeed there are extremely rare cases where an operator had to step in for another for the final acts of some sub-order. However, we shall assume in the following that each order is associated to a single picker — if two are present in the original database, we shall only consider the first, as they always pick almost all of the items of the order in any case.
- `P/P` is the number of items found in a *package* as received from the manufacturer and stocked in the warehouse. The total number of pieces `PCS` is invariably a multiple of this value.
- `LOCATION` is an identifier for the picking position where the items were stored at the time that this act took place; it encodes aisle–bay–side information as well as which picking position and module the SKU was found in.
- `DIST` is the distance, in metres, that a picker would ideally have to have moved to get from the start of their path to the picking position of the act.
- `START` and `END` encode the instant at which data for the act associated with the row was read and confirmed by the picker

SKU	PCWT	...
33	0.05	...
76	0.5	...
196	0.2	...
4341	0.1	...
⋮	⋮	...

LOCATION	X	Y	Z	SIDE
03-02-L-1	13.0	4.8	0.0	L
03-05-R-3	11.0	13.6	0.0	R
09-33-L-2	37.0	80.0	0.7	L
10-20-L-1	41.0	48.8	0.0	R
⋮	⋮	⋮	⋮	⋮

Table 3.2: Example of item and position database data.

and the instant at which the picker scanned all items after placing them on the roller, respectively. These are usually stored as date–time pairs with a resolution at least as fine as seconds; Table 3.1 shows only times and not dates for brevity.

It must be noted that from this dataset one cannot derive which sub-orders come from the same overall order. While this does not have any relevance for prediction — we are interested in predicting the behavior of a *single* picker working on a *single* roller — it is important for the optimization model, as it does not and cannot know what a complete order would have looked like in the old warehouse.

The example in Table 3.1 also shows some of the main issues of the dataset: looking at the two rows shown for order 196, and assuming that rows are sorted by order and start time, one act ends at 21:45:33 and the next begins at 21:46:50; this is an example of what was mentioned in Section 2.2, i. e. that pickers may forget to scan items after placing them on the roller, or equivalently may fail to confirm they began the next act. Not only does this lead to sudden "jumps" in order fulfilling, but it also misrepresents the time taken by acts before and after it: the second act of the aforementioned pair begins at 21:46:50 and, according to that row in isolation, it would appear that operator USER2 had traveled a distance of 9.4 metres and picked two items of SKU 691 in the span of a single second — a description that clearly cannot be reflective of what actually occurred at the time.

`LOCATION` information is taken from here as opposed to the item database (see Section 3.1.2) due to the fact that orders span multiple months, during which many SKUs switched places; the item database records their position in the moment at which it was exported. The same is true for `P/P`, though changes in the number of pieces per package are not controlled by the warehouse and were only relevant for a handful of SKUs.

3.1.2 Item database

To enable use of more data pertaining to each SKU one may refer to the warehouse's *item database*: a table detailing salient features for each

item, such as their physical dimensions, weight, fragility, turnover frequency, their latest known position in the warehouse, et cetera.

The item database is intended to contain such information for each SKUs present in the order reports as per Section 3.1.1; however, as it only contains data for items that are present in the warehouse at a given point in time, it may not have all data pertaining to SKUs that have progressively stopped being stocked. Indeed, for the case at hand, two such databases were required for the dataset: one of them contained data for all SKUs stocked in the warehouse approximately one month after the data in Section 3.1.1 was collected, and another from a sister warehouse, i. e. a warehouse servicing the same business, which stocks a greater variety of SKUs. The latter was required because, as mentioned, our order data comprised 24 weeks, during which some items stopped being stocked, and thus the second database was sourced to add the missing information for those items and complete the picture.

Unfortunately, during this process it was found that data for many items that appeared in both databases was inconsistent; for example, volume was measured for a single item in one database and for the whole package in the other, and the two used fragility measures that did not always agree with one another.

The left side of Table 3.2 shows the only data point that was consistent between the two databases. There is one row per SKU, whose code, corresponding to the codes present in column `SKU` for each act of the order reports (Section 3.1.1), is stored in column `SKU` of the item database. Column `PCWT` contains the value of that datapoint, which is the net weight in kilograms of a single item pertaining to that SKU. This value allowed us to fill in missing entries for `WT`, which, as mentioned above, could be incorrectly set to zero (see Section 3.2).

3.1.3 Position database

While location data as given in the order report data may be useful by itself if one is looking at a single warehouse, it does not provide us with information over the spatial disposition of shelves; this is even more the case in the old warehouse where data was collected, as some aisles belong to different areas (see Section 2.1.2) and so an incremental aisle number, for example, would not provide sufficient information to derive its actual position.

This data is obtained from a *position database* such as the one on the right side of Table 3.2. Each `LOCATION` identifier is unique and relates to a single picking position; these correspond to the identifiers present for each act in the `LOCATION` column of the order reports (Section 3.1.1). For each position, columns `X`, `Y` and `Z` describe its three-dimensional Cartesian coordinates, in metres, with respect to a coordinate system shared between all picking positions of the same

zone, as per Section 2.1.2. The coordinate systems for each zone only differ in their origin point, which was not given to us; the `X` and `Y` coordinates point in the same direction for each zone and `Z` is always the vertical coordinate component — that is, it is the height of the given picking position from the floor of the warehouse. Furthermore, for our case all aisles of each area were oriented so that they either ran parallel to the `X` axis or to the `Y` axis.

The last column present in the position database is `SIDE`. As the picker moves through the warehouse in a fixed path and they traverse the aisles in only one direction, when they reach any position in their route it will always consistenly be either to ther left or to their right. `SIDE` accordingly contains the value "R" for right and "L" for left for each position. Integrating the coordinates `X` and `Y` with the column `DIST` of the order reports, as well as the fact that aisles are parallel to either of those two axes as mentioned above, it is possible to derive the value of `SIDE`, and thus this column is technically redundant. However, as this would require some effort as well as knowledge of the structure of the warehouse (which we intend the final dataset to be independent of, as described in Section 3.3), this column was nevertheless included in our data.

We also note that the inconsistent nature of coordinates across the warehouse does not pose a problem for our purposes, as we consider each order separately, and any one of them will only pertain to one and only one zone; no picker will "cross" a coordinate boundary at any point.

3.2 DATA CLEANING

The first step in handling raw data is to remove inconsistencies and remove outliers or unnecessary data from it. The most easily dispatched such issues were as follows:

- Some values of `SKU` do not refer to actual items stored in the warehouse; they instead encode other activities of the warehouse as if they were items. As these are not actually handled by pickers, and seeing as they form their own subset of orders that encompasses only "SKUs" of this sort, all such acts can be safely filtered out.

- As mentioned in Section 3.1.1 it is sometimes the case that `WT` is set to zero for some SKUs; this is easily fixed by cross-referencing with `PCWT` in the item database. After this step `WT` is never zero for any act.

After these operations we replace `PCS`, the total number of items, with `PKGS`, the total number of *packages*; since, as mentioned in Section 3.1.1, `PCS` is always a multiple of `P/P`, it is enough to divide the former by

the latter to obtain this new value. Since pickers place entire packages onto the roller as opposed to single items, this provides a more realistic description.

The rest of this section is dedicated to those issues that required greater care in their handling.

3.2.1 Duplicate act aggregation

At times, the WMS will hand "duplicate" requests to pickers, that is, ask them to pick a certain amount of some SKU after they already did so. As the WMS handles this automatically pickers never have to retrace their steps for this; both acts will then appear one after the other in our data.

This is of course not a valid representation of how pickers actually behave: as they are already at the given SKU's picking position, they will treat the two (or more) requests as a single act. From the point of view of our data, what this means is that all duplicate acts after the first will take an unnaturally short amount of time, as it is the only kind of act where pickers are not required to walk to a new destination. In many cases, in fact, the time between `START` and `END` was under one second and was rounded down to *zero*.

To prevent such outliers from appearing in the finalized dataset we *aggregated* all duplicate acts in the following manner:

1. Acts are partitioned by `ORDER` and sorted by `START` time and, secondarily, `END` time.

2. In each partition we assign to each act an `INDEX`; it is 0 at the start of each order and increments whenever SKUs of subsequent acts differ. Duplicate acts will be assigned the same `INDEX`.

3. Acts with the same `ORDER` and `INDEX` are merged together into a single act. `WEIGHT` and `PCS` of the merged act equal the sum of all duplicates, and `START` and `END` equal the earliest and latest of the duplicates, respectively. The remaining columns are the same between duplicates and are kept as is.

Note that sorting by `END` time is significant, as the step is meant to group duplicate acts together. Suppose a duplicate act starts at time t and is listed as ending at time t also, whereas the next act, for a different item entirely, also starts at time t as would be expected. Sorting solely by `START` may lead to the latter occurring earlier in the act sequence than the duplicate, which would break this assumption.

3.2.2 Start and end time fixing

As explained in Section 3.1.1 acts are supposed to end as soon as the next one begins; however, this is very often not the case. It would also

not have been reasonable to consider such acts outliers: a significant fraction of the orders contained such a discrepancy (though often small and only measuring a few seconds) and it would not have been reasonable to filter out all of them. It is nevertheless possible to remove the very few orders that feature a *large* time difference: such inconsistencies occur almost invariably either for the last act of an order — which, we assume, the picker forgot to log — or around times when pickers would take breaks and leave the picking area mid-order. We chose 10 minutes as a cut-off point; the condition was then met by approximately 0.43% of all orders.

Our choice was to fix act times by *averaging* the `END` time of each act with the `START` time of the one following it in the same order, and use the result as the new value for both. For pairs of acts that are already adjacent in time this has no effect at all, whereas for small time differences — on the order of a few seconds — timing for both acts, while unlikely to represent actual warehouse events, is more in line with other acts of similar length and weight than before. What is more, most cases of unnaturally short acts that had not been fixed by aggregation are here lengthened to a more reasonable value.

At the end of this step we computed for each act its `TIME`, that is, the difference between `END` and `START`, and dropped any further reference or use of `END`.

3.3 ANONYMIZATION AND WAREHOUSE INDEPENDENCE

As previously mentioned in order to make the dataset viable for use in training models that can be applied to more than one warehouse — in our case, the same warehouse at different points in time — and with a changing workforce, it is necessary that the data in output not have any reference to either picker identity or to structural details of the warehouse. In this section we deal with both issues.

It must be noted that one step towards warehouse independence was already taken through use of the item database; if other features of it had been available, it would have been at this point that they would have been joined to the order report. In so doing, the final dataset only refers to properties of SKUs and not their identity, allowing reuse of models trained on it for new, previously unseen SKUs.

3.3.1 Picker feature extraction

As any mention of which orders were handled by which picker must not be retained in the output, we strove to obtain data from it that could provide anonymous picker-dependent information about orders. To do so we assigned each order for a given picker to a *shift*: it is

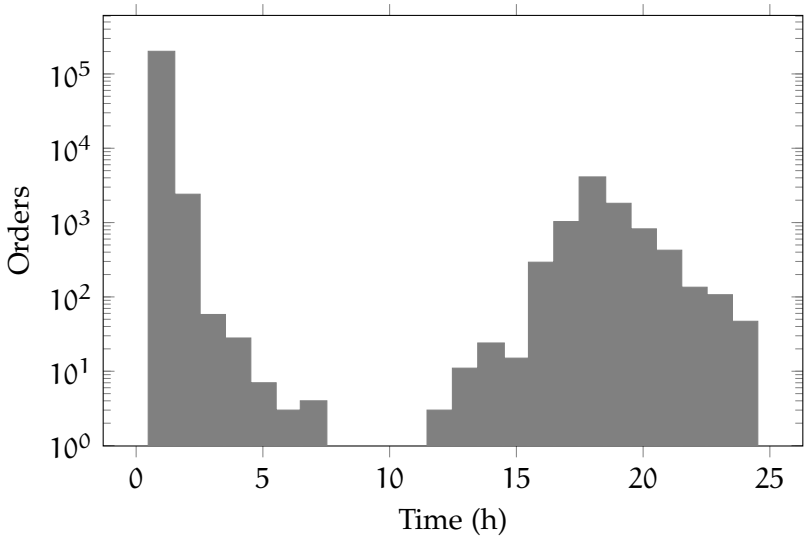

Figure 3.1: Order-to-order time difference histogram.

assumed that sufficiently long breaks between orders for a given picker are indicative of them leaving the warehouse and returning the following day or later on.

In particular, for each order we compute the earliest `START` of any act inside of it. Orders for each `PICKER` are sorted and each is assigned its own `SHIFT` as a temporary column; its value is 0 at the start and increments whenever the earliest `START` time for two successive orders is greater than a given time difference. For our case, as seen in Figure 3.1, there is a clear division between differences under approximately 6 hours and above; thus this value was chosen as the cut-off point.

The pair of `PICKER` and `SHIFT` identifies one continuous period of work for each picker. Several performance indicators related to *fatigue* can be extracted from this data; our choice was to compute, for each order, the total weight `TOTWT` picked on all previous orders of the same shift. Total picked weight per day is a common fatigue metric in logistics, and so it was extended to each shift, "truncating" the workday to just prior the order for which this metric is to be computed.

It is of course possible to compute other kinds of features, for example exerted work — weight times distance traveled with it for all previous orders — or the time offset between the start of an order and that of its shift. The former was not used as it is dependent on the structure of the warehouse, specifically on the total length of the picking path; the latter does not directly provide information related to fatigue, as it does not specify in any way how much work was done during that time. Furthermore, its encoding, using the same technique as in Section 3.4.1, would require much additional space.

In either case, `PICKER` is ultimately removed from the dataset; this removes any information that directly identifies a picker; `SHIFT` is also

removed as it does not offer any information that is in itself useful for prediction, as it is merely a progressive counter used to identify shifts. Thus the only remaining column is — for our chosen feature — `TOTWT`, the total picked weight before the start of the given order.

3.3.2 Position data

Adding positional data to remove opaque location information is a straightforward process: for each act, `LOCATION` is replaced by the columns in the position database seen in Section 3.1.3. Additionally, for the three columns that describe to horizontal movement, namely `X`, `Y` and `DIST`, the differences between each successive act were computed and stored in `DX`, `DY` and `DDIST` respectively. This allows each act entry (row of the dataset) to display all movement information for the picker in a self-contained manner.

In most cases, `DDIST` is a positive value; however, for some cases — which are relevant for picker behavior and cannot be readily filtered out — pickers must backtrack and return to a picking position they had already passed through as no items may have been available at ground floor at the time. From the picker's perspective, however, the distance traveled, which is what `DDIST` tracks, is the same; it is only the *direction* of travel that differs. To model this `DDIST` was replaced by its absolute value, and its sign was spun off as its own column `DIR`.

We note here that, unlike the other two anonymization steps, this replacement is only of use for generalizing datasets when applying them to separate warehouses. For a single warehouse structured similarly to the ideal one seen in Figure 2.1, for example, it would have been just as effective to encode aisle–bay–side as their own abstract coordinates, thus obviating the need for a separate position database.

3.4 NUMERICAL REPRESENTATION

This final step deals with two main concerns: how to encode processed data to a purely numerical form that is useful for our purposes and how to store it concisely and efficiently. The first concern deals with three issues; two can be dealt with immediately, whereas the third — how to encode `START` — is handled further below.

On one hand there is still a component of the data that is encoded as a label, namely `SIDE`. We choose to convert it to a numerical value such that L is stored as -1 and R as 1; this conveniently leads to a feature whose statistical distribution has average close to 0.

On another, it would not be reasonable to provide `WT` in the output as had been previously handled: for very small items picked individually its value could be as low as 0.01, i. e. ten grams, whereas very large orders — for example, large cuts of meat — would easily run into the

tens of kilograms. Thus we chose to encode weight as its *logarithm*. This choice also represents the way weight is felt by the picker, in the sense that it magnifies weight differences between light items — those light enough to be picked single-handedly — and "compresses" those between heavy items, which instead require significant effort to lift into the roller.

For consistency the same encoding was employed for `PKGS`, as it and `WT` are linearly related for all SKUs. We do not, however, encode the total previous weight as described in Section 3.3.1 as a logarithm; this is not only because the total weight would be zero for the first order of any shift, but also because the field is intended to track picker fatigue, which is approximately linearly related to the total weight.

For the second concern we chose to split our feature set between *order-level* features and *act-level* features:

- Order-level features are those extracted from a picker's identity as per Section 3.3.1 as well as the `START` time for the *first* act of the order.
- Act-level features comprise everything else: `WT`, `DIST`, `PKGS`, all positional data and `TIME` for each act.

Each order is stored as a pair: an order-level vector and an act-level sequence of vectors, encoded as a table (a rank-2 tensor).

Note that we do not output `START` for *all* acts, but only for the first, i. e. the beginning of order processing. This is for two reasons. One is economy of representation: the step in Section 3.2.2 ensures that `START` for any subsequent act can be obtained from the first one by adding the preceding `TIME`s, and so is redundant. The other, which is of great importance for our purposes, is that for the model presented in Chapter 4 to work properly in an inference scenario — which is exactly what is required of it in the optimization step — one would have to recompute the representation of `START` as described below after inferring each act; such usage would be computationally expensive on top of its redundancy making it unnecessary.

3.4.1 Time encoding

Much like `SIDE` above, it is desirable to convert `START` into a purely numerical form, as opposed to the date-time in the raw data. One immediate property of the specific data available to us was that, as it consisted of only 24 weeks, not all months of the year were featured: it would thus have been pointless to provide date information in the output, as a major portion of the input space would have been missing. Thus, as an initial step, each value of `START` was converted into a fraction of the time of day t in the range $[0, 1[$, with 0 corresponding to midnight.

To represent time so that any ML process that uses it may properly recognize not just the instant at which an event takes place but also time differences between two events, one must note that time is inherently cyclical; it is then important to represent it in such a way that any two instants that are close to each other in the input also be numerically close in the output representation.

Consider, by way of example, encoding time as a simple fraction t as above. in many cases, the distance in time between two instants would be simply the difference between the two fractions; a time of $\frac{1}{2}$ — noon — is close to $\frac{11}{24}$ — 11 AM. However, times near midnight and just before midnight would have been very close in the input, but far from each other in the output representation: a time of 0 — midnight — is far from $\frac{23}{24}$ — 11 PM. Thus this encoding must be ruled out.

In practice, a common way to represent such data is the *circular representation*: each value of the cyclical data is mapped onto a circle and what is stored are the coordinates of the resulting point, i. e. the sine and cosine of the angle. For our purposes, this means replacing the fraction t with the two "coordinates" $t_x = \cos(2\pi t)$ and $t_y = \sin(2\pi t)$. This encoding, while efficient — both in computation time and in memory, as it only requires one additional value for each instant with respect to encoding just the fraction t — and properly representing the input in the above sense, has the issue that identifying whether an instant is in a given time range is less than trivial. For example, if it was useful for the ML process to identify whether an order began between 8 AM and 10 AM, one would require two simultaneous conditions on t_x and t_y, namely $t_x \leqslant 0$ and $\frac{1}{2} \leqslant t_y \leqslant \frac{1}{2}\sqrt{3}$ — but the structure of these conditions is heavily dependent on the exact range.

We therefore chose a similar approach that makes the latter problem easier and also allows us to represent time data with an arbitrary number n of values, much like the coordinates of the circular representation above, where increasing n affords greater precision in handling time ranges. This representation is built on *radial basis functions* — functions that depend only on the distance between the input point and an arbitrary given point. In particular, we chose the following *Gaussian radial basis function* adapted for the circular case:

$$\phi_k(t) = \exp\left(-4\left(\left(\left(nt - k + \frac{n}{2}\right) \bmod n\right) - \frac{n}{2}\right)^2 \log 2\right),$$

where the $a \bmod b$ operation is assumed to be defined exactly as $a - b\left\lfloor \frac{a}{b} \right\rfloor$ and k is the index of the function as a natural number in the range 0 to $n-1$.

The values of $\phi_k(t)$ are shown in Figure 3.2 for $n = 6$. This family of functions is periodic in the range $t \in [0, 1]$ and each $\phi_k(t)$ is radial with respect to the point $\frac{k}{n}$: the value of the function $\phi_3(t)$ in the figure, for example, depends only on the distance between t and $\frac{1}{2}$, that is, $\phi_3\left(\frac{1}{2} + x\right) = \phi_3\left(\frac{1}{2} - x\right)$ for all $0 \leqslant x \leqslant \frac{1}{2}$; this is true

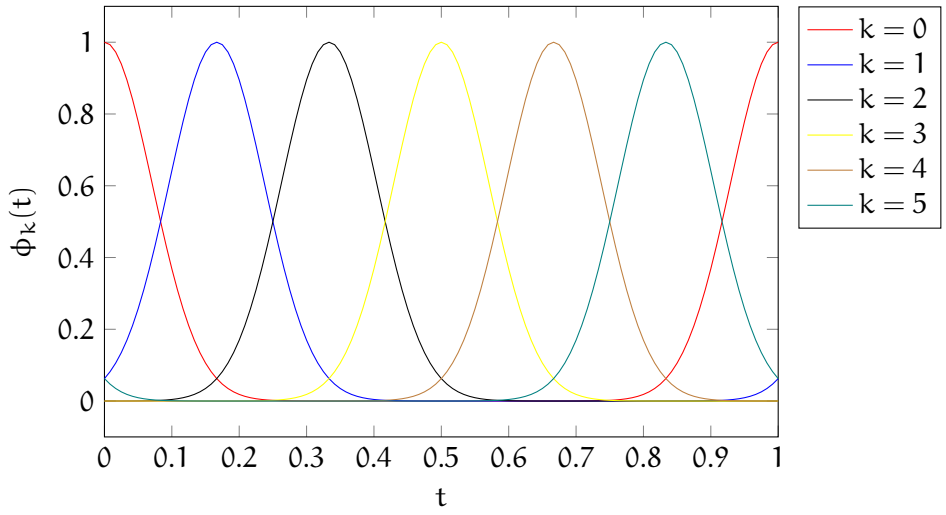

Figure 3.2: Gaussian radial basis functions for $n = 6$.

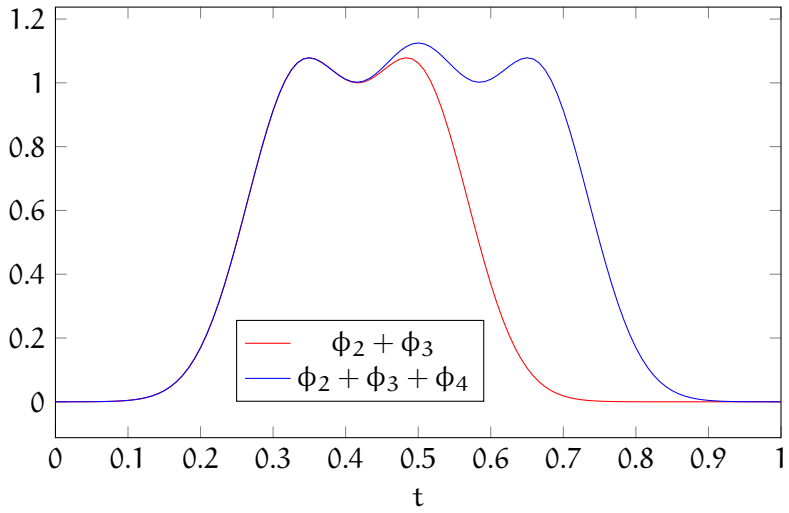

Figure 3.3: Sum of adjacent Gaussian radial basis functions.

of all $\phi_k(t)$ for all n, with the caveat that the distance is measured "wrapping around" between 0 and 1, as is clear from the figure.

As may ben seen both from its definition and from the graph above, these functions have the shape of a Gaussian function with values in $[0, 1]$ and maximum at $\frac{k}{n}$, made periodic by use of a mod operator. The exact form of $\phi_k(t)$ was chosen so that any two functions ϕ_k and ϕ_{k+1} for $k < n - 1$, as well as ϕ_0 and ϕ_{n-1}, would have value $\frac{1}{2}$ at the "mid-way point" $\frac{1}{n}\left(k + \frac{1}{2}\right)$, and thus for each t except for the midway points there is exactly one k for which $\phi_k(t) \geqslant \frac{1}{2}$.

To justify this choice, as well as the claim of being able to identify values in a range more easily, we show what a sum of "adjacent" functions (i. e. functions with adjacent k values) in Figure 3.3. The choice of a mid-way point value of $\frac{1}{2}$ is convenient in that it can be computed efficiently — as the exponential in the definition of

$\phi_k(t)$ becomes a power of two — and also ensures that all values of the sum between the peaks of the first and the last function are at least 1. Indeed, the example of $\phi_2(t) + \phi_3(t)$ is seen to be a good approximation of the indicator function for the range $\left[\frac{1}{3}, \frac{1}{2}\right]$, and similarly for other sums; thus, for a large class of ranges, it is enough to check if a sum of adjacent functions — or a weighted sum for ranges that end between the peaks of two adjacent functions — is greater or lower than a given value, and all such conditions have a similar form.

The final representation is computed by replacing the fraction of time of day t with $t_k = \phi_k(t)$ and storing each t_k as its own column `STARTk`, replacing column `START`. The chosen number of divisions for our case was $n = 6$, providing a distance of 4 hours between each adjacent function's peak, which was considered reasonable for our purposes.

4 PREDICTION MODEL

This chapter focuses on the problem of predicting picking time from order data and on an approach based on a RNN architecture for its regression.

To the author's knowledge, the only prior studies known for this problem in the field of warehouses are by Kabak and Güleç, 2022 and Balzereit, Soni and Bunte, 2023, though the more general problem of modeling and predicting travel time from route information and past behavior occurs often in the literature (see e. g. Wang, Fu and Ye, 2018). Indeed, previous analyses of picking time focused on obtaining mathematical models of the same, or their statistical properties, in the context of idealized warehouses with known picker behavior and layout (Le-Duc and de Koster, 2007; Pan, Wu and Chang, 2014). This is due to the fact that this problem has only found relevance by its applications to optimization of order batching or location assignment, but not by itself.

4.1 PROBLEM STATEMENT

Let R be a random variable spanning *order-level data*, modeling the real-life distribution of the same; this variable represents the probability distribution of all data pertaining to an order as a whole, such as its start time, picker-related data, as well as the number of acts that make up the order. This last value is denoted $n(R)$, as it is a deterministic function of R; for brevity we shall refer to this value as just n.

Furthermore, let $A_1, \ldots, A_n$ be a sequence of random variables similarly spanning *act-level data*, jointly distributed with respect to each other and R; each A_i represents the probability distribution of all data pertaining to a single act of an order, such as the picking position of the items, the total number of items picked, their weight and so on, with the length of the sequence being the function $n = n(R)$ as mentioned above and the acts A_i sorted chronologically, i. e. as the picker does them.

Finally, let T be another random variable with values in $\mathbb{R}^+$, assumed to be also jointly distributed with respect to R and $A_1, \ldots, A_n$, where T given $R, A_1, \ldots, A_n$ models the time taken to service the order described by R and $A_1, \ldots, A_n$.

The problem of *pick time prediction* is then that of finding a family of distributions $f\left(t \middle| \vec{\theta}\right)$ with parameters $\vec{\theta}$ as well as a computable

estimator $\hat{\theta}$ of those parameters such that the latter *minimizes* the *relative entropy* between the distribution of T (denoted τ) and f

$$\hat{\theta} = \arg\min_{\vec{\theta}} D_{KB}\left(\tau \,\middle\|\, f\left(\cdot \,\middle|\, \vec{\theta}\,(R, A_1, \ldots A_n)\right)\right) \tag{4.1}$$

where the relative entropy (or *Kullback-Leibler divergence*), whose general form for distributions $P(x)$ and $Q(x)$ with $x \in X$ is

$$D_{KB}(P \parallel Q) = \int_{x \in X} P(x) \log \frac{P(x)}{Q(x)}\, dx,$$

is taken over the full distribution of $(R, A_1, \ldots, A_n, T)$. This divergence is used as it measures the expected loss of information between the ground truth distribution of τ and the predicted distribution f considered as its approximation, hence its name.

It is clear, however, that neither τ nor the distribution of any of the other random variables is available to us. It is nevertheless still possible to find an approximate solution to the problem by estimating such distributions with a sufficient amount of past order data. For that, let $\mathcal{R}$ be a multiset of values of $(R, A_1, \ldots, A_n)$; the approximate distribution will then have probability $\frac{1}{|\mathcal{R}|}$ for each element, and the minimization problem will take the form

$$\hat{\theta} = \arg\min_{\vec{\theta}} \sum_{(r, a_1, \ldots, a_n, t) \in \mathcal{R}} \frac{1}{|\mathcal{R}|} \log \frac{1}{|\mathcal{R}| f\left(t \,\middle|\, \vec{\theta}\,(r, a_1, \ldots, a_n)\right)}$$

where, as $|\mathcal{R}|$ is constant with respect to $\vec{\theta}$, the right side reduces to

$$\begin{aligned}\hat{\theta} &= \arg\min_{\vec{\theta}} \sum_{(r, a_1, \ldots, a_n, t) \in \mathcal{R}} -\frac{1}{|\mathcal{R}|} \log f\left(t \,\middle|\, \vec{\theta}\,(r, a_1, \ldots, a_n)\right) \\ &= \arg\min_{\vec{\theta}} \mathbb{E}\left[-\log f\left(T' \,\middle|\, \vec{\theta}\,(R', A_1', \ldots, A_n')\right)\right]\end{aligned}$$

where the primed variables are distributed as the approximation through $|\mathcal{R}|$. This approach is termed *empirical risk minimization* (Vapnik, 1991) and leads to an optimization problem that can be handled by common techniques for *supervised learning* in the context of regression: we are tasked with minimizing the expectation over dataset $\mathcal{R}$ of a function

$$\mathcal{L}\left(t \,\middle|\, \vec{\theta}\right) = -\log f\left(t \,\middle|\, \vec{\theta}\right) \tag{4.2}$$

of the *label* t and of the value of the $\vec{\theta}$. This function is called the problem's *loss*, and its form is that of a *negative log-likelihood* — it measures the information content of t, i. e. how unexpected such a time is in the context of the output distribution.

Note that T is not a deterministic function of order data; indeed, even for a single act there is great variability in the time that a picker takes

to fulfill it. Thus, while it is possible to construct a similar problem that focuses on, say, computing a point estimate $\hat{t}$ for time and minimizing the absolute error of the prediction, this would provide an inaccurate picture of the problem: when applying these results to warehouse planning, time differences are still significant, and this is in no small part due to factors beyond what can be learned from order data.

An alternative formulation of the problem, which is what we will focus on in the following, is that of predicting not so much the total order picking time T but rather picking time for each act $T_1, \dots, T_n$ separately; this can be readily obtained from the previous problem, assuming times for each act are known, by augmenting the dataset with all "prefixes" of all orders. From the dataset (i. e. the approximate distributions) point of view it is enough to replace $\mathcal{R}$ with

$$\mathcal{R}_P = \bigcup_{(r,a_1,\dots,a_n)\in\mathcal{R}} \{(r[m], a_1, \dots, a_m, t_m) : 1 \leqslant m \leqslant n\},$$

where $r[m]$ is the same as r save for $n(r) = m$ and t_m is the time spent picking the mth act, which as previously stated is assumed known. Thus the predictor is now tasked with estimating the distribution of t_m given all acts for that order up to the mth, and as this can be repeated for any $1 \leqslant m \leqslant n$, we may obtain all estimates for all acts as desired.

Similar modifications are required to fit this new formulation to the problem stated for exact distributions as in Equation 4.1: $R, A_1, \dots, A_n$ must be replaced with $R[m], A_1, \dots, A_m$ for all m, where $R[m]$ is a random variable that represents the same distribution as R for the first m acts of all orders with at least m acts (i. e. $n(R) \geqslant m$) and $A_1, \dots, A_m$ have the same joint distribution with $R[m]$ that they had with R in the original formulation; furthermore, the time to be estimated is not the total for the order T but rather T_m, i. e. the time spent picking the last act given. The structure of the problem remains otherwise the same and the same considerations as for the approximate distributions apply.

This alternative does lead to issues when the dataset is not properly fit for it (see e. g. Section 3.2.2), but it has multiple advantages as well: first of all, it limits over-fitting of the solution to the dataset — in our case, inability to generalize to other warehouses of similar size and layout — since a greater amount of datapoints become available. More importantly, it allows use of the model to predict which acts in an order take the longest, which can be of use in contexts other than our case of storage allocation.

4.2 NETWORK STRUCTURE

We introduce here the main structural components of our network, looking at their features through the lens of supervised learning.

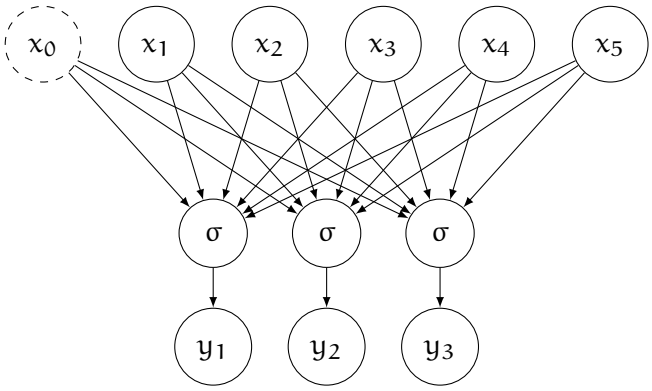

Figure 4.1: Structure of a neural network layer.

4.2.1 Deep neural network

Neural networks are networks composed of computational cells called *artificial neurons* that seek to emulate, in an approximate sense, how a biological neuron is affected by external stimuli. Each neuron is connected to an arbitrary number of inputs and computes a single value as its output in the following manner:

$$y = \sigma\left(w_0 + \sum_{1 \leqslant i \leqslant n} w_i x_i\right)$$

where x_i are its inputs, w_i are the *weights* according to which inputs are linearly combined, and σ is a function called the neuron's *activation*. Weights for each neuron are collectively called the network's *parameters*, as they are the only values that are determined during the training process.

Figure 4.1 shows a simple network with five inputs and three neurons (outputs). For brevity it is assumed there to exist a "constant" input $x_0 = 1$, so that the final computation of outputs from inputs may take this form: $\vec{y} = \sigma(W\vec{x})$, with all inputs and outputs collected into a vector, all weights into the matrix W, and σ acting elementwise. We shall also assume this convention in the following. It is also generally the case, as in the figure, that all neurons share the same activation function.

Neural networks are commonly organized in a sequence of *layers*; neurons of each layer take as input either the whole network's inputs or the previous layer's output. Referring back to Figure 4.1, $\vec{y}$ could serve as the $\vec{x}$ vector for the next layer, or, if the last layer, as the output of the entire network. Layers intermediate between network inputs and outputs do not have their values directly perceived by users of the network and are thus called *hidden layers*; the number of layers of a network is termed its *depth*, hence networks with at least one hidden layer are called *deep*.

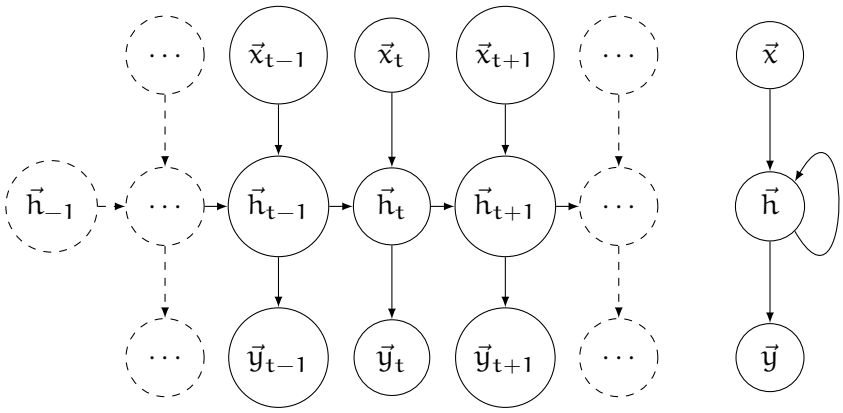

Figure 4.2: Schematic structure of a RNN.

This structure, termed *feed-forward* due to layers feeding their results "forward" (i. e. towards outputs), was originally devised to emulate the brain's layered structure, and it was assumed that such an arrangement would lead to later layers learning more "complex" representations of the input. While compelling by itself, Hornik, Stinchcombe and White, 1989 later proved that even a deep network with a single hidden layer can approximate, when furnished with sufficient neurons and given suitable activation, any (measurable) function to any accuracy desired — in other words, they are universal approximators.

In order to train the parameters of a network by far the most common method is that of *gradient descent* through *back-propagation*. Gradient descent assumes the ability to compute partial derivatives $\partial_w \mathcal{L}$ of the loss with respect to any of the parameters w of the network, and after that "descend" in the direction of the gradient by updating weight w to $w - \eta \partial_w \mathcal{L}$ for some pre-determined η called the *learning rate*; for sufficiently small values of η this process moves the weights towards local minima for $\mathcal{L}$. Back-propagation allows for the efficient computation of such derivatives by automatic differentiation for each layer (i. e. of its outputs as a function of its weights and inputs), and computing the derivative for $\mathcal{L}$ by applying the chain rule "backwards" through the network; efficiency comes from reuse of derivatives for each layer when computing those for the previous one.

4.2.2 Recurrent neural networks

RNNs are a specialized class of deep neural networks that are able to process sequences of data of *arbitrary* length: this is accomplished by means of a *hidden state* $\vec{h}$ which is passed from one computation step to the other. Instead of feeding the network the entire sequence at once, each element is processed one after the other along with the previous state, and the next state is computed along with the output of the network.

A schematic view of a RNN is shown in Figure 4.2. On the right we have the architecture as described in its *folded* form above; the left side more accurately depicts data dependencies between separate time steps i. e. the steps in the input sequence. In its most general form a RNN processes its input and generates its output with two functions, namely the *update* function $\vec{u}$ and the *output* function $\vec{o}$:

$$\vec{h}_{t+1} = \vec{u}\left(\vec{h}_t, \vec{x}_t\right) \quad \vec{y}_t = \vec{o}\left(\vec{h}_t\right).$$

For simple RNNs the update function is the same affine transformation with elementwise activation as feed-forward networks and the output function is the identity; this is not the case for more complex architectures. We also note that, as each time step requires a previous state as input, so does the first time step: in that case a special initial state $\vec{h}_{-1}$ is assumed to exist; it is usually an all-zero vector.

One key advantage of RNNs is *parameter sharing*: all of the values of $\vec{h}_t$ and of $\vec{y}_t$ in Figure 4.2 are computed in the same way, with the same functions having the same parameters, for all elements of the sequence. Furthermore, each output provably depends only on previous inputs and not on future inputs: this is in contrast to ordinary feed-forward networks trained on the same sequences (e. g. by padding all training sequences to some maximum length) that have to learn weights for all elements of the sequence separately.

The output of such a network may be either the sequence $\vec{y}_0 \dots \vec{y}_{N-1}$, and therefore the effect of the network is transforming a sequence into another, or just the final value $\vec{y}_{N-1}$, in which case the network is used to summarize or obtain a representation of the entire sequence. As RNNs can be used as an intermediate layer in a deeper network, it must be noted that only RNNs are "aware" of the sequential nature of the data being processed; parameter sharing is extended to the entire network, and so any non-recurrent upstream or downstream layer will only process the output of the previous layer.

As a result of this, while the same back-propagation technique can be applied to RNNs, the partial derivative of the loss at time t with respect to any weight of the RNN or of any layer prior to it will expand to include, due to $\vec{u}$, the same weight used at all preceding time steps. In other words, gradient computation will take place as if the network were unfolded. This extension of back-propagation is called back-propagation through time (BPTT) (Werbos, 1990); since every time step depends on the partial derivatives of the previous, training such a network is expensive, as derivatives must be stored (and made use of) for all steps of the sequence.

Unfortunately, one issue that became readily apparent when working with simple RNNs was the following: suppose that we have such a network with *linear* activation $\sigma(x) = x$ and zero initial state, and let

$$\vec{u}\left(\vec{x}, \vec{h}\right) = U\vec{x} + W\vec{h},$$

with both U and W trainable weight matrices. Due to parameter sharing the complete unfolded equation for $\vec{h}_t$ becomes

$$\vec{h}_t = \sum_{0 \leqslant \tau \leqslant t} W^{t-\tau} U \vec{x}_\tau,$$

and the weight matrix W that encodes state-to-state connections appears to powers up to t. Any eigenvalues λ of W will have their counterpart in W^t as λ^t; if $|\lambda| \gg 1$, then the component of $U\vec{x}_i$ along its eigenvector will increase exponentially each time step, and oppositewise if $|\lambda| \ll 1$. Hence, unless λ is close to unity, information will either be quickly lost in successive steps or amplified to the point of numerical instability.

This issue is known as *vanishing* and *exploding gradients* (Bengio, Simard and Frasconi, 1994) due to its effect on BPTT — powers of the same matrices also appear in the computation of gradients, which causes instability and poor performance while training. While multiple solutions to this problem are known, the most effective way is to change $\vec{u}$.

4.2.3 Gated recurrent units

Long short-term memory (LSTM) units (i. e. cells — pairs of $\vec{u}$ and $\vec{o}$) were introduced in Hochreiter and Schmidhuber, 1997 precisely to solve the vanishing and exploding gradient problems, or rather their downstream effects on detecting long-term patterns; their far superior performance compared to simple models based on simple RNN made them the standard recurrent architecture. GRUs were introduced later in Cho et al., 2014 as a simplification of LSTM cells; Chung et al., 2014 shows they have similar properties to the latter units while requiring fewer parameters. This early study in their effectiveness provides a clue that these units may be useful for our case, as they dealt with the similar problem of sequence modeling, i. e. predicting the next element of a sequence when given all preceding elements.

The internal structure of a single GRU cell is shown in Figure 4.3. $\vec{h}_t$ is the state of the hidden layer at time t and $\vec{x}$ is the input to the layer. All cells contain two *gates*, that is, values to be computed that sit in the range $[0, 1]$ and are multiplied with other values to form the neural network equivalent of an on-off switch. These are the *reset gate*

$$\vec{r} = \sigma\left(W_r \vec{x} + U_r \vec{h}_{t-1}\right)$$

and the *update gate*

$$\vec{z} = \sigma\left(W_z \vec{x} + U_z \vec{h}_{t-1}\right),$$

where in both cases σ is the logistic sigmoid function applied elementwise. Following that, a temporary new value for the hidden state is

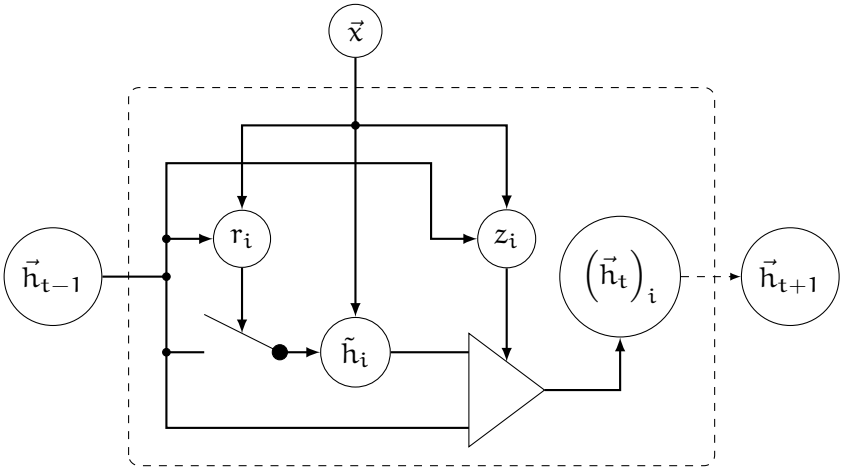

Figure 4.3: Internal structure of a GRU cell.

computed, which makes use of the reset gate to "choose" whether or not to include the previous hidden state:

$$\tilde{h} = \phi\left(W\vec{x} + U\left(\vec{r} \odot \vec{h}_{t-1}\right)\right)$$

where ϕ is any activation function, most commonly tanh, and $\odot$ is elementwise multiplication. Finally, the hidden state at time t is computed as a linear interpolation between its previous state and the current temporary value:

$$\vec{h}_t = \left(\vec{1} - \vec{z}\right) \odot \vec{h}_{t-1} + \vec{z} \odot \tilde{h}$$

where $\vec{1}$ is a vector of all ones. The initial state of this RNN is $\vec{h}_{-1} = \vec{0}$. Note that gates $\vec{r}$ and $\vec{z}$, as well as the temporary value $\tilde{h}$, do not form part of the hidden state; while they ultimately participate in the computation of $\vec{h}_t$, they are not themselves connected to the next time step.

When the reset gate is close to 0 its respective hidden cell does not contribute to $\tilde{h}$, nor therefore to $\vec{h}_t$, and so it is treated as if the network began processing a new sequence, hence the name. Similarly, when the update gate is close to 0 its respective hidden cell will have value $\left(\vec{h}_t\right)_i \approx \left(\vec{h}_{t-1}\right)_i$, that is, it will not be updated. This ability of the network to maintain state and update it with a pattern that may depend on inputs and hidden states is what allows it to bypass the disastrous effects of vanishing and exploding gradients on long-distance information retention.

4.2.4 Overall structure

A schema of our network is presented in Figure 4.4. Its computation proceeds as follows:

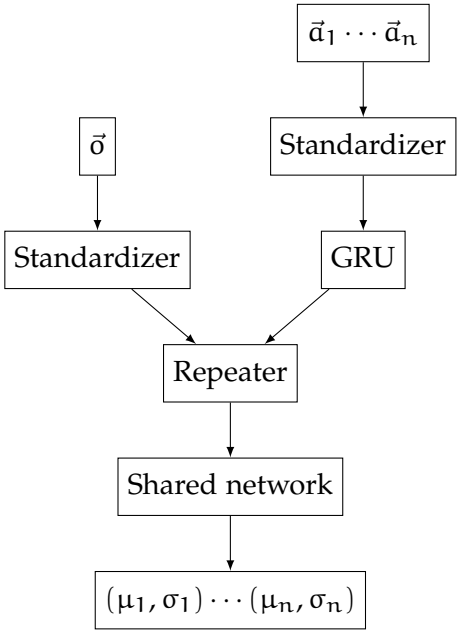

Figure 4.4: Structure of the prediction network.

1. Input data is split, as before, between order-level $\vec{o}$ and act-level $\vec{a}_1, \ldots, \vec{a}_n$.
2. All elements are then *standardized*; this is done by computing the mean μ and variance σ for each element of $\vec{o}$ and of each $\vec{a}_i$ and returning $o'_i = \frac{o_i - \mu}{\sigma}$ (and similarly for all $\vec{a}_i$).
3. $\vec{a}'_1, \ldots, a'_n$ are passed to a GRU layer, which outputs values $\vec{y}_1, \ldots, \vec{y}_n$.
4. $\vec{o}'$ is concatenated to each step of the output of the GRU layer by use of a repeater element; the sequence becomes $(\vec{y}_1, \vec{o}), \ldots, (\vec{y}_n, \vec{o})$.
5. Finally, the concatenated vectors pass through a shared feed-forward network, which outputs distribution parameters for each time step.

The choice of the output distribution was made after analysis of picking acts with the same `DDISTANCE`, a sample of which for the case of position-to-position movement between acts is shown in Figure 4.5. Multiple distributions in common use for modeling wait times that would fit the data in were considered, namely the *gamma distribution*

$$f(x \mid \alpha, \theta) = \frac{1}{\Gamma(\alpha)} \theta^{-\alpha} x^{\alpha-1} \exp\left(-\frac{x}{\theta}\right),$$

the *Fisk* or *log-logistic distribution*

$$f(x \mid \alpha, \beta) = \frac{\beta}{\alpha} \left(1 + \left(\frac{x}{\alpha}\right)^{\beta}\right)^{-2} \left(\frac{x}{\alpha}\right)^{\beta-1},$$

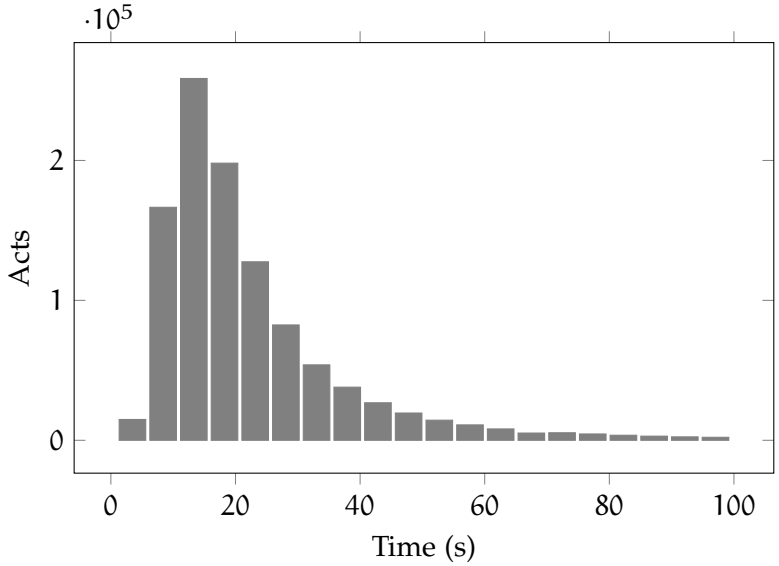

Figure 4.5: Order time histogram for acts between adjacent positions.

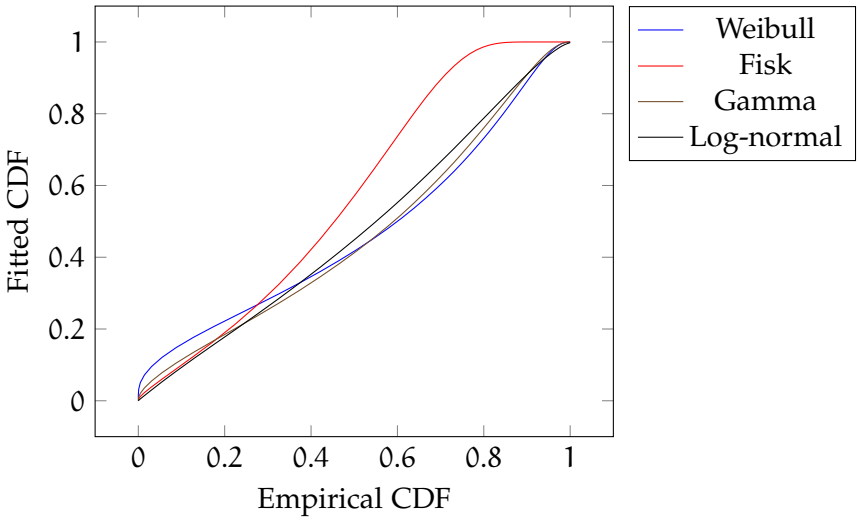

Figure 4.6: P-P plot for acts of Figure 4.5.

the *Weibull distribution*

$$f(x \mid \beta, k) = \beta k (\beta x)^{k-1} \exp -(\beta x)^k,$$

and the *log-normal distribution*

$$f(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma x} \exp\left(-\frac{1}{2}\left(\frac{\log x - \mu}{\sigma}\right)^2\right).$$

Figure 4.6 shows a P-P plot — a plot of the cumulative distribution for a MLE fit of the theoretical distribution over the empirical cumulative distribution — of the same acts as before for the above distributions. As goodness-of-fit is measured by how close each distribution is to the bisector $x = y$, it is clear from the figure that the log-normal nearly perfectly captures the statistics of the dataset; we therefore chose it as our output distribution $f(t \mid \mu, \sigma)$.

The use of a RNN architecture allows the network to learn a representation of the input sequence up to each time-step t through its hidden states $\vec{h}_t$, and from this representation predict times for each act. Thus the model itself is able to learn what sort of information should be propagated from one act to the next directly; a model that instead predicts picking times looking at the data for each act in isolation would instead require careful *feature engineering*, i. e. devising which features should be computed ahead of time and included in act-level data, such as, for example, the total roller weight at that point in the order. Such features can of course still be included for a RNN architecture, and will not worsen prediction performance if they are, but as the model is constructed so as to learn a representation of its own there is less of a need for them. Furthermore, the number of such features that one could imagine may impact picking time is quite large, and including all of them would not be feasible, nor would testing which ones improve prediction performance; thus we preferred to use an approach that automatically learns a suitable representation.

4.3 TRAINING

As already mentioned in Section 4.1, training of the model assumes a negative log-likelihood loss as per Equation 4.2, and in particular that of the log-normal distribution

$$\mathcal{L}(t \mid \mu, \sigma) = \log t + \log \sigma + \frac{1}{2}\left(\frac{t - \mu}{\sigma}\right)^2. \tag{4.3}$$

The activation function for GRU gates and output are those of the original paper, namely the logistic sigmoid $\sigma(x) = \frac{1}{1+\exp(-x)}$ and tanh, whereas all standard hidden feed-forward layers use tanh as activation. For the output layer μ is obtained by linear combination — i. e. identity

Inputs: dataset $\mathcal{R}$, epochs E, batch size b, decay rate ρ, learning rate η

for $i \leftarrow 1$ **to** E **do**
  split dataset $\mathcal{R}$ into batches of size b
  $s \leftarrow 0$
  foreach batch B **do**
    $g \leftarrow 0$
    foreach sample $(r, a_1, \ldots, a_n, t) \in B$ **do**
      $g \leftarrow g + \frac{1}{b} \left(\nabla_w \mathcal{L}\left(t \mid \theta\left(r, a_1, \ldots, a_n\right)\right)\right)$
    end
    $s \leftarrow \rho s + (1 - \rho) g^2$
    $w \leftarrow w - \eta \frac{1}{\sqrt{\epsilon + s}} g$
  end
end

Algorithm 4.1: The RMSprop algorithm in the context of our network.

activation — whereas σ, which is required to be positive, is computed via the *softplus* function $s(x) = \log\left(1 + \exp x\right)$.

Generation of the initial weights for each layer followed Glorot and Bengio, 2010 by taking each weight of the shared network and each weight for the W matrices of the GRU cells from a uniform distribution

$$w \sim \mathcal{U}\left(\left[-\sqrt{\frac{6}{I + O}}, \sqrt{\frac{6}{I + O}}\right]\right),$$

where I and O are the number of inputs and outputs of the layer, respectively. The U matrices of the GRU cells were instead initialized with random orthogonal matrices obtained by QR decomposition of a random matrix with normally-distributed entries, as described in Saxe, McClelland and Ganguli, 2014. All biases were initialized to zero.

Computing the gradient for gradient descent was done, as is standard, by sampling the dataset in *batches* (in our case, of 128 orders each) and computing the average with respect to each batch. This is an estimate of the expectation of the loss that significantly speeds up training at minimal cost, if any, in the performance of the results. Specifically, we use the RMSprop algorithm (Hinton, 2012) as shown in Algorithm 4.1; due to the relatively large size of the dataset and the speed at which the network was found to converge, as well as its lack of overfitting, more sophisticated adaptive gradient descent algorithms such as Adam (Kingma and Ba, 2015) did not provide significant difference in performance.

The algorithm works as follows: for each batch the average gradient of $\mathcal{L}$ as a function of all weights is computed and stored in g, and a rolling average s of the square of the gradient is maintained for each

	Total	Training	Validation	Testing
Orders	221,453	155,033	46,494	19,926
Acts	6,958,977	4,876,440	1,454,179	628,358

Table 4.1: Order and act counts for the dataset.

weight. The average decays with a rate given by ρ — by default kept constant during training at 0.9. Then, prior to computing the new set of weights by gradient descent with learning rate η, the gradient is rescaled by a factor equal to $\frac{1}{\sqrt{s}}$, with an added ε (machine epsilon) to stabilize this factor for very low values of s. This is repeated for the entire dataset for a given number of *epochs* E. As claimed by the original developers of this algorithm, it allows training to get stuck in areas of the weight space that provide similar loss less often; it is in common use for RNN training and was found to provide adequate training speed.

70% of the dataset was used for training, with the remaining 30% split further into 70% used for validation for network parameters and evaluation of fitting ability and 30% left for testing. Care was taken to only adapt the standardization layers to the training set so as to not contaminate the training stage with data from the validation step. The training dataset was also shuffled during each training epoch.

4.4 RESULTS

The dataset obtained after the pre-processing step was comprised of 221453 orders for a total of 6958977 acts, and as mentioned in Section 4.3, 70% of it was used for training, 21% of it was used for validation, and 9% was used for computing all prediction results shown; this last split is traditionally known as the *testing* split. Order and act counts for each split are shown in Table 4.1. The average number of acts per order was therefore about 31.42.

The data for each order of the dataset was split in the following three categories, where the names correspond to the data generated by the pre-processing of Chapter 3:

- Order-level data comprised 7 values per order, namely encoded start times `STARTk` for $0 \leqslant k \leqslant 5$ and the fatigue metric `TOTWT`.
- Act-level data comprised 11 values per order, namely the total weight picked for the act `WEIGHT`, the number of packages picked `WQU`, the distance traveled to get to the picking position `DISTANCE`, the aisle side `SIDE` of the picking position, coordinates `X`, `Y` and `Z`, and finally their differences with respect to the previous act `DX`, `DY`, as well as the distance traveled during the act `DDISTANCE` and its direction with respect to the picking route `DIR`.

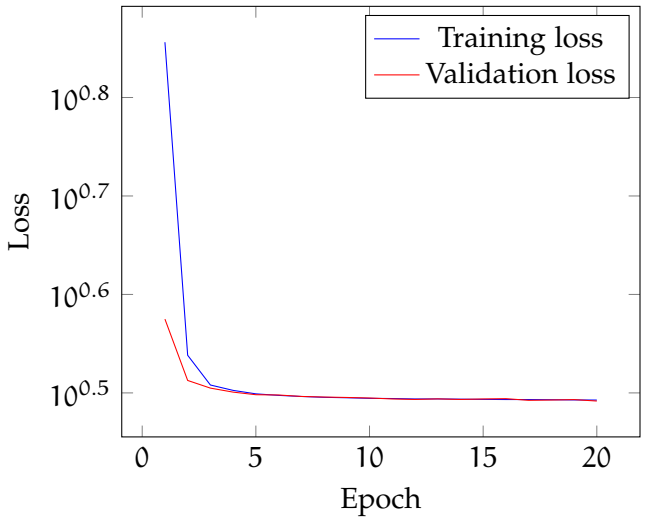

Figure 4.7: Loss and validation loss for 32 units with $\eta = 10^{-4}$.

- *Labels* for each act, i. e. their picking time `TIME`.

We also note for the benefit of later analysis that the average picking time in the dataset was 36.361.

The loss function we used is, as described in above in Section 4.3, the negative log-likelihood of the log-normal distribution as in Equation 4.3, where where the term $\frac{1}{\sqrt{2\pi}}$ of the log-normal distribution was dropped as it is constant and does not influence the minimization process of $\mathcal{L}$. We note that with this convention the theoretical lowest value that can be attained by $\mathcal{L}$ would be $\log\sqrt{2\pi} \approx 0.9189$.

The network was trained with a shared network consisting of a single hidden layer of 32 neurons and a GRU layer of a variable amount of neurons as a power of two: $n \in \{8, 16, 32\}$. Similarly, learning rate was allowed to vary in powers of ten among $\eta \in \{10^{-2}, 10^{-3}, 10^{-4}\}$. The RMSprop algorithm was used for training and its decay rate was set to the default value of $\rho = 0.9$. The limited number of choices for these parameters was due to the long training times required by the size of the training set.

Lower values of n than the ones mentioned above performed much worse than the values reported here, whereas for $n > 32$ we did not detect any progress with respect to the case $n = 32$. Similarly, for $\eta < 10^{-4}$ the training process converged more slowly to similar values of the prediction loss as those shown in Figure 4.7; for large values $\eta > 10^{-2}$, on the other hand, the training process was not able to converge at all: this is expected, as any training algorithm that makes use of a fixed learning rate, such as Algorithm 4.1, adjusts the weights of the prediction model based on the learning rate after each batch; if the learning rate is too large, every batch will have too great an effect on the weights, and the bias inherent in estimating $\mathcal{L}$ for each batch will not lead to convergence of the overall model.

	η		
n	10^{-2}	10^{-3}	10^{-4}
8	3.09346	3.10259	3.11488
16	3.08627	3.09104	3.11563
32	3.07633	3.08795	3.10301

Table 4.2: Validation loss matrix for different hyperparameters.

An example of the results of most trained models is seen in Figure 4.7. Training loss was computed as the average negative log likelihood over the last batch of $b = 128$ orders used while training during that epoch. Similarly, the validation loss values were computed as the average of the same loss function $\mathcal{L}$ over the *entire* validation set. For both, the values of μ and σ were generated by the prediction model after it completed its epoch of training.

Even for relatively low values of η training appears to plateau after only a few epochs; this is most likely due to the large amount of data available at each epoch, as the speed of convergence varies along the training–validation split. Equivalently, the dataset is sufficiently general so that even a few passes over it already lead training towards a local minimum.

Quickly-decreasing loss and validation loss are generally seen as positive, especially when the value it converges to is either very low for the chosen loss function or when it is strongly dependent on *hyperparameters*, i. e. parameters that describe the learning process as opposed to the network's connections. Table 4.2, however, shows that our training process did not meet either condition.

The network was trained using the same random seed — thus generating the same initial conditions and the same batches for each epoch — with several pairs of hyperparameters, and the validation loss was computed, as above, is an average of $\mathcal{L}$ over the entire validation dataset. The results are essentially constant, save for the slight increase as η decreases and n increases. This is believed to be caused by the inability of the network to adapt to pickers moving between separate areas in the same zone, that is, going from one group of aisles to another, with the two possibly having differing orientation.

As an example of this, we picked a random order from the test set; this order was comprised of 25 acts, of which 23 had the picker move in the same area, and 2 had the picker move between different areas. For two of the former and the two latter acts we ran the prediction model and drew the resulting distribution in Figure 4.8 and Figure 4.9, respectively. For all output distributions the mean of the distribution was also computed, shown as a dashed green line, and all were annotated with their "ground truth" value — that is, the time as found in the test dataset — shown as a point labeled GT.

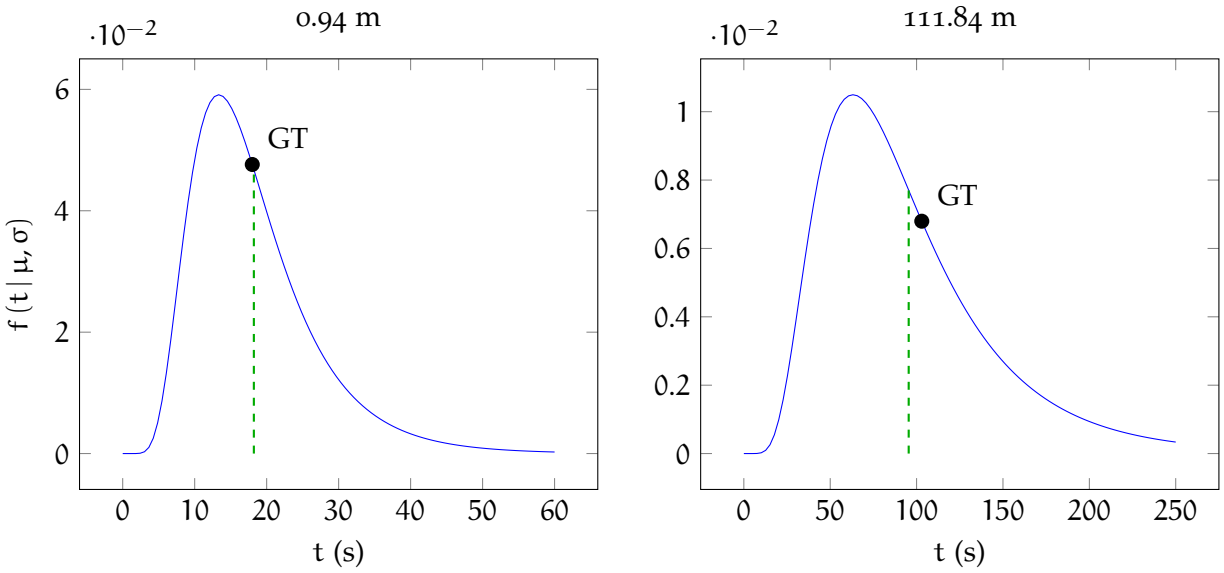

Figure 4.8: Predictions for intra-area travel distances.

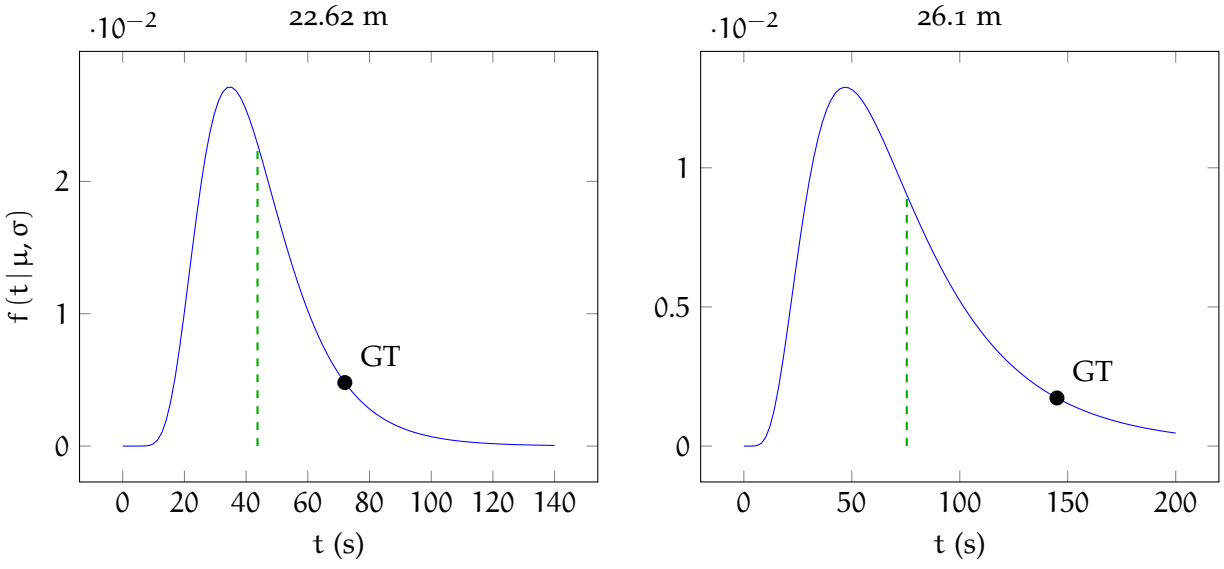

Figure 4.9: Predictions for inter-area travel distances.

The results of Figure 4.8 for for the two intra-aisle acts show how the predictor generally behaves with respect to them. In principle, as we seek to minimize $\mathcal{L}$ one would expect the predictor to learn that the value of the distribution $f(t \mid \mu, \sigma)$ should maximize its value around the ground truth time t, and therefore one would expect to find that a good fit for the dataset would have its ground truth values crowd around the maximum — the *mode* — of the predicted distribution. Such a model would however not have truly predicted the distribution of picking times, as it would be nearly the same as a simpler predictor that outputs only the mode, and for a model of complexity similar to ours it would not be reasonable to expect predictions so exact for even a significant fraction of picking times.

On the other hand, we see from the output of our model that good predictions have t crowd around the *mean* of the distribution instead; this observation holds true even for strikingly long distances, such as over 100 m as seen in the second graph, as long as the picker does not leave the area. Such a result is interesting as it is not a direct consequence of minimizing $\mathcal{L}$, as explained above. This may be explained as follows: suppose that possible picking times for a specific act are more or less evenly distributed around a relatively large interval; to minimize expected $\mathcal{L}$ one would then need to find not so much the point t at which $f(t \mid \mu, \sigma)$ is maximum, but rather the bounds of an interval that maximizes the probability of drawing a point contained in it from the same distribution. For large intervals it may be noted directly from inspection of the shape of the log-normal distribution that this is an interval centred near the mean of the distribution; in this sense we may thus say that the model captures the actual distribution well, as opposed to predicting a single point.

Inter-area picker travel, by contrast, is not handled well by the model at all; as Figure 4.9 shows, the prediction severely undershoots the actual time taken. It is by and large this kind of error that prevents the loss function from being optimized to a lower value: as predictions for high picking times naturally have higher σ, the distribution around the ground truth will have a lower value of $f(t|\mu, \sigma)$ and consequently higher negative log-likelihood. As the model is unable to capture this behavior even in an approximate sense, and as GT is specifically higher than expected from the mean, it will find itself in a region where the distribution is particularly low, namely the tail end of the log-normal as in the left graph in Figure 4.9.

It must also be noted that the training dataset contained only 135249 acts where pickers moved between one area and another; this accounts for only 2.77% of the acts used for training. It is then not entirely unexpected for the model not to have been able to learn to estimate their distribution, as the time spent traveling between separate areas is distributed much more sparsely and there were all in all far fewer examples to learn from than for intra-area travel, for which the loss

function could be minimized more effectively. Thus the model undershoots time for these acts, predicting them to take time closer to that of similar intra-area acts rather than their actual time.

Nevertheless, the prediction error for intra-area acts is approximately 22.3%, measured as absolute deviation from the mean of the output distribution. We conclude that the model fails to capture the complex structure of the old warehouse, but is still well-suited for predicting order times of the more regularly-structured new warehouse, as it is composed of a single area.

5 OPTIMIZATION MODEL

In this chapter we tackle the problem of finding an optimal placement for each currently stocked SKU in the new warehouse, with constraints on the order in which items are to be picked and on their possible placements in the warehouse. To do so we employ a metaheuristic-based approach with both soft and hard constraints coupled with random search.

As claimed by Islam and Uddin, 2023, the literature on this problem is dominated by approaches that describe it as an integer programming optimization problem and then solve it, in an approximate sense, either by some ad-hoc heuristic — for simpler formulations — or by metaheuristics. Such approaches include minimizing travel distance between pairs of SKUs that appear often together in orders via a greedy initial approximation followed by either a genetic algorithm (GA) or simulated annealing (SA) (Li, Moghaddam and Nof, 2016; Wei Jiang and Wang, 2021; Yuan, Wang and Li, 2019; Zhang, Wang and Pan, 2019); directly minimizing cost of warehouse handling — modeled as a linear function — using similar evolutionary algorithms or local search (Guerriero et al., 2013; Muppani (Muppant) and Adil, 2008; Wang, Zheng and Wang, 2011); and a similar travel distance minimization approeach with SA (Yuan et al., 2021).

Two notable cases are that of Lee, Chung and Yoon, 2020, who simultaneously minimize travel time as well as congestion through a multi-objective evolutionary algorithm, and Bottani et al., 2012, whose analyzed warehouse is similar to that of this work, as it considers picking constraints based on item fragility and solves the problem using a GA. The multi-objective approach has also been employed with other sets of goals, e. g. in Jiao et al., 2018 and in Fontana et al., 2020; Atmaca and Ozturk, 2013 sum multiple goals together into a single function to minimize and then apply SA to it.

Despite much previous research, the joint problem of time estimation and minimization is not tackled as often. Daria Battini and Sgarbossa, 2015 propose a multinomial approximation for travel distance then use it to optimize storage locations; however, their framework has not, to our knowledge, been improved upon in subsequent papers, and efforts have been made to either apply their approach to other warehouse structures or to replace the approximation with other heuristics. Thus we consider the problem at hand largely unexplored.

5.1 PROBLEM STATEMENT

Let $\mathcal{P}$ be the set of picking positions in the warehouse, and let $\mathcal{I}$ be the set of SKUs that must be stored in it; a *location assignment* is an injective function $c : \mathcal{I} \to \mathcal{P}$.

The unconstrained *storage location assignment problem* is the task of finding an assignment c that, in general, minimizes some performance indicator of the warehouse; in our case, we consider expected travel time and thus write

$$\hat{c} = \arg\min_{c \in S} \mathbb{E}\left[T \mid c\right]$$

where T is, as in Section 4.1, a random variable modeling the time taken to pick a random order $R, A_1, \ldots, A_n$, and $S \subseteq \mathcal{P}^{\mathcal{I}}$ is the search space for assignments — presently that of injective functions. The expectation of T is taken, as shown in the equation above, with the assumption that the warehouse is structured as in c.

The form of this problem is that of a *combinatorial optimization* problem — the space of possible solutions S is finite (and discrete) — however, much like pick time prediction, it is not solvable as is, since the distribution of T is not known. A similar approach is then posited for estimating the expectation, namely the existence of a dataset $\mathcal{R}$ of past orders, whence an approximate distribution of $R', A'_1, \ldots, A'_n$ is derived. Assuming further the existence of a predictor t for picking time given order data, the above minimization problem becomes

$$\hat{c} = \arg\min_{c \in S} \sum_{(r, a_1, \ldots, a_n) \in \mathcal{R}} \frac{1}{|\mathcal{R}|} t\left(r, c\left(a_1, \ldots, a_n\right)\right),$$

where $c\left(a_1, \ldots, a_n\right)$ indicates that position data for dataset acts is given by the assignment c — including the final ordering of such acts according to the fixed, assignment-independent picking route. Such a predictor may be obtained, for example, by computing the time required to move from one picking position to the next ahead of time through statistical analysis of past order data and summing these values for all acts, or, as in our case, by predicting the distribution of T itself and computing its expectation given all the order-level and act-level information at our disposal.

While the unconstrained problem is interesting in itself, a further restriction is often required in practice; this is due to there being *constraints* posed on the configuration. We consider here two such kinds:

- *positional constraints*, i. e. limiting which position–SKU pairs are allowed; this is modeled by a function $P : \mathcal{I} \to 2^{\mathcal{P}}$ that assigns to each SKU $i \in \mathcal{I}$ a subset of positions it may be placed in, and require that all $c \in S$ satisfy $c(i) \in P(i)$ for all $i \in \mathcal{I}$.

- *roller constraints*, which require that any order picked in the warehouse have its items placed on the roller, i. e. picked, in a certain order; this is modeled by a total preorder relation $\precsim$ between SKUs, where $i \precsim j$ if i *may* be picked before j. The condition on assignments c may then be described in terms of the mapping $\gamma(c(a_1, \ldots, a_n))$ from an order's acts $a_1, \ldots, a_n$ to the picking sequence of its items; in particular, if $(i_1, \ldots, i_n) = \gamma(c(a_1, \ldots, a_n))$ then

  $$\forall 1 \leqslant k \leqslant l \leqslant n. \quad i_k \precsim i_l;$$

  that is, the sequence is preordered with respect to $\precsim$.

It is also often the case that some constraints are *soft*; that is, not all must necessarily hold for a solution to the problem, but it is still desirable for all of them to hold. We follow here the most common method of adding such constraints to an optimization problem, namely a generic *penalty* function added to the optimization goal: for positional constraints this will be a function $\pi_P(c)$ of the configuration c, irrespective of orders, whereas for roller constraints it will be a function $\pi_R(\gamma_c(a_1, \ldots, a_n))$ that penalizes each act according to how far away it is from the ideal case of being fully preordered.

All in all, the constrained storage location assignment problem takes the following form:

$$\hat{c} = \arg\min_{c \in S'} (\pi_P(c) + \mathbb{E}[T + \pi_R(\gamma(c(A_1, \ldots, A_n))) \mid c])$$

with its own associated expectation estimate being

$$\hat{c} = \arg\min_{c \in S} \left(\pi_P(c) + \sum_{(r,\alpha) \in \mathcal{R}} \frac{1}{|\mathcal{R}|} (t(r, c(\alpha)) + \pi_R(\gamma(c(\alpha)))) \right), \quad (5.1)$$

where in the interest of brevity we denoted $\alpha = (a_1, \ldots, a_n)$.

5.2 METAHEURISTICS

As mentioned in the introduction to this chapter, it is known that this problem is known to be NP-hard even in the simpler, linearly-modeled case, hence the use of heuristics and search strategies for near-optimal solutions instead of global optima. Our formulation, which includes a predictor t — regarded here as a black box, i. e. without its internals known to the optimization algorithm — cannot be solved exactly either; we thus make use of two *metaheuristics* to guide a local, random search process.

5.2.1 Simulated annealing

Simulated annealing (SA) is an algorithm inspired by statistical mechanics and in particular by metalworking; it seeks to simulate the way that particles in a metal behave when subjected to cooling (Kirkpatrick, Gelatt and Vecchi, 1983). Each step of the algorithm is given a *temperature* T, which is lowered in subsequent steps through use of a *cooling strategy*; for the annealing process of metals, initial temperature T_0 is quite high, reflecting that of molten metal. The most common cooling schedules are those employed in the original paper, i. e. the linear schedule, which decreases temperature after each step by a constant value, and the exponential schedule, which multiplies temperature by a constant $0 < \kappa < 1$ called the *cooling rate*.

Particles move randomly through the search space attempting to find a state (a solution) with the lowest energy (the function to be minimized). At each step, if a state of lower or equal energy is found, it is immediately accepted as the new state; if the randomly-generated state is instead at a higher energy value, it may still be accepted, though only with a given probability $p(\Delta E, T)$ which is a function of the (positive) energy difference between the initial and the generated state and the temperature at that step. This probability is almost invariably obtained by the formula $p(\Delta E, T) = \exp\left(-\frac{\Delta E}{kT}\right)$, where k is a parameter of the algorithm called its *Boltzmann constant* in reference to the physical constant underlying similar processes in nature. Allowing the solution to move to a "neighbor" that is worse than itself is a strategy that is meant to prevent the algorithm from finding a local minimum of the function too early in the search process.

While SA is commonly used in the storage assignment literature, as per the introduction to this chapter, we shall use in the following a variant of it known as *threshold accepting* (Dueck and Scheuer, 1990). It is a simplification of standard SA and was shown to be optimal — in the sense of the energy of the final configuration found — within a large class of algorithms that move randomly through a finite search space (Franz, Hoffmann and Salamon, 2001).

Threshold accepting is shown in Algorithm 5.1. The algorithm works as follows: an initial configuration c is generated randomly, and at each of r steps of the algorithm it is modified in a random manner (i. e. it is *mutated*, by analogy with the terminology of the next metaheuristic) through a given mutate operation to give a new configuration c'. The energy e, i. e. the function to be minimized, is computed for both c and c'; c' is accepted as the new solution if the difference ΔE is below the temperature t, as opposed to the probabilistic acceptance of SA. Finally, temperature is cooled through an exponential cooling schedule with rate κ. The output configuration c_{best} is the best configuration found during the entire execution of the algorithm.

The main insight underlying this variant is that the random mutate operation is sufficient to escape local minima, and therefore it is

Inputs: rounds r, initial temperature T_0, cooling rate κ
Output: best configuration found c_{best}
$t \leftarrow T_0$
$c \leftarrow$ random
$c_{best} \leftarrow c$
for $i \leftarrow 1$ **to** r **do**
    $c' \leftarrow \text{mutate}(c)$
    if $e(c') - e(c) < t$ **then** $c \leftarrow c'$ **end**
    $t \leftarrow \kappa t$
    if $e(c) < e(c_{best})$ **then** $c_{best} \leftarrow c$ **end**
end
return c_{best}

Algorithm 5.1: The threshold accepting SA variant.

enough to provide a *threshold* for steps that increase energy — that being the temperature — rather than a probability. On the other hand, as this threshold is directly responsible for the trajectory that configurations c take through the search space, it is even more important that it be correctly dimensioned for the values of energy that $e(c)$ can take; if too large, initial steps will be effectively random, and if too small, the algorithm may get stuck on a local minimum.

5.2.2 Genetic algorithm

Genetic algorithms (GAs) are a class of algorithms that take inspiration from the natural process of evolution, considered as a system through which individuals of a population seek to maximize their fitness with respect to the environment in a sequence of successive generations (Holland, 1992). Solutions to the problem are encoded in a suitable format; these are called the *chromosomes* of an individual, each composed of individual *genes* that can be modified independently; multiple individuals, each with their own chromosome, make up a population.

The purpose of such an encoding is so that, at each generation, individuals are able to "mate" with each other — that is, chromosome recombination is simulated in a way that is sensible for the problem at hand. The generated offspring is then mutated, again much like genes mutate in nature, and some of the individuals that make up the current generation are replaced by some of the offspring, generally according to an evaluation of the *fitness* of each individual — this being the function to be maximized by the algorithm.

Inputs: generations g, (initial) population size μ, offspring λ
Output: best individual found c_{best}
$p \leftarrow \{\text{random}, \ldots, \text{random}\}$ where $|p| = \mu$
$c_{best} \leftarrow \arg\min_{c \in p} e(c)$
for $i \leftarrow 1$ **to** g **do**
    $\{(a_1, b_1), \ldots, (a_\lambda, b_\lambda)\} \leftarrow \text{parents}(p)$
    for $j \leftarrow 1$ **to** λ **do**
        $c_j \leftarrow \text{crossover}(a_j, b_j)$
        $c_j \leftarrow \text{mutate}(c_j)$
    end
    $p \leftarrow \text{replace}(p \cup \{c_1, \ldots, c_\lambda\})$
    if $\min_{c \in p} e(c) < e(c_{best})$ **then**
        $c_{best} \leftarrow \arg\min_{c \in p} e(c)$
    end
end
return c_{best}

Algorithm 5.2: Generic GA.

As described, the algorithm structure, shown in Algorithm 5.2, is extremely general. Aside from the fixed size of the population μ and the fixed amount of offspring at each generation λ, the entire process is described in terms of functions called *evolutionary operators*; the output of the algorithm is the best configuration c_{best}, i. e. the most fit individual, found in any of the populations p generated during the execution of the algorithm.

parents and replace are both *selection* operators that pick a subset of individuals from a larger population, for the purpose of generating pairs of parents or deciding which individuals will "survive" and be passed on to the next generation, respectively. These operators will generally depend on the fitness function but are otherwise independent from the representation or the problem itself, and as a result a wealth of such operators have been proposed in the literature.

The crossover operator is used to "merge" two individuals into one — the offspring — in a randomized manner, emulating the way that chromosomes cross over in nature. Some operators choose to maintain any structure on the chosen representation by construction of the offspring, whereas others employ simpler merging operations and then apply *repair* techniques to re-establish the aforementioned structure.

The mutate operator, much like the random search for SA, is intended to modify a single solution so that the whole search space can

be reached by any individual; without it, the only chromosomes that will
random population via crossover. In general, mutation operators act on each gene of the chromosome independently in a randomized manner.

The main strength of GA is the diverse set of solutions that are part of each generation. Indeed, much of the work in developing a successful set of operators is in maintaining the balance between *exploration*, i. e. the ability to search as much of the space S of solutions as possible, and *exploitation*, i. e. further improving optimal solutions: exploration benefits from mutation and crossover operators that modify the parents as much as possible and provide varied offspring and selection operators that allow even less fit solutions to be a part of the "reproduction" process, whereas exploitation is precisely the opposite.

5.3 EVALUATION FUNCTION

The *evaluation function* $e(c)$ is the function that both SA and GA minimize. For the constrained location assignment problem as described in Equation 5.1, this function depends on the definition of three other functions: the time estimator t and the two penalty functions π_P and π_R for soft positional and roller constraints, respectively.

In order to estimate time we use the predictor t as described and trained in Chapter 4 and compute the expected value of the distribution it generates as its prediction for each act. Both for it and for π_R a dataset of past orders $\mathcal{R}$ is required to compute the expected order picking time; each order in it will have its position data for each act replaced with that given by the configuration c under evaluation, and its sequence of acts will be sorted according to the picking route in the warehouse. These two processing steps are denoted as the single operation $c(a_1, \ldots, a_n)$ in Section 5.1 and we were supplied with a position database for the new warehouse (with the same structure as the one in Section 3.1.3) to compute position data as was done in Section 3.3.2.

The full dataset for the old warehouse, denoted here as $\mathcal{R}_F$, was used as a starting point; this is the same dataset that underwent preprocessing in Chapter 3 as well as that used for training, validation and testing of the predictor. As can be read from Table 4.1, this dataset is comprised of several millions of acts, and as a result, computing t as well as $c(a_1, \ldots, a_n)$ for each configuration c to be evaluated at each SA round or GA generation for all $(r, a_1, \ldots, a_n) \in \mathcal{R}_F$ was unfortunately not computationally feasible.

To cope with this we chose to estimate $e(c)$ by *sampling* from the dataset $\mathcal{R}_F$ in the following manner:

1. A sample $\mathcal{R}$ of ν orders is drawn from $\mathcal{R}_F$ uniformly at random without replacement.
2. $\mathcal{R}$ is used as a dataset to compute the expected values of t and π_R as per Equation 5.1 for the following b rounds of SA or generations of GA.
3. The sample $\mathcal{R}$ is re-drawn after b rounds or generations have elapsed.

The behavior of this sampling process thus depends on two parameters: the number of orders per sample ν and the number of rounds or generations that use the same sample b.

One further issue is that computing t for a given configuration c using the sample $\mathcal{R}$ drawn with the method above leads to very large variability in the average predicted order picking time, especially for low sample sizes ν (e. g. under $\frac{1}{1000}|\mathcal{R}_F|$). This is in part to be expected as it is merely an estimate of the average value of t over $\mathcal{R}_F$; still, a significant part of this variability is due to the fact that for such low sample sizes the statistical distribution of the number of acts per order in the sample $\mathcal{R}$ is unlikely to be a good approximation for the same in $\mathcal{R}_F$, and total picking time for an order strongly depends on the number of acts it is composed of.

We thus chose to eliminate part of this variability by computing the average picking time not per *order* but rather per *act* of all orders in the sample $\mathcal{R}$; the form of $e(c)$ as seen in Equation 5.1 does not change, as this only concerns the definition of t, and since the predictor described in Chapter 4 also predicts picking times for each act, its implementation is straightforward. Minimizing $e(c)$ with this t still minimizes average order picking time, but as the number of acts per sample $\mathcal{R}$ is necessarily much larger than the number of orders, average act picking time is less variable on a sample-by-sample basis.

As for the penalty functions, our case did not have soft positional constraints, and therefore no analysis was undertaken as how to best encode them as a value fit for π_P. On the other hand, roller constraints were present, and thus a method is needed to compute π_R such that minimizing it will also lead to as many orders as possible having their acts sorted with respect to the preorder $\precsim$.

Two methods have been identified as candidates for π_R. One modifies the computation for the time estimate component t by reordering acts inside each order using $\precsim$ and then computing $e(c)$ with that ordering. This leads to an increase in the estimated time, and the difference between it and the unconstrained estimate would be the definition of π_R for this case. Such a formulation involves no additional parameters and models the behavior of a picker who would forgo the requirement of strictly adhering to the picking route in favor of picking items in the optimal order according to roller constraints; however, it requires that the predictor be able to cope with acts that move in the

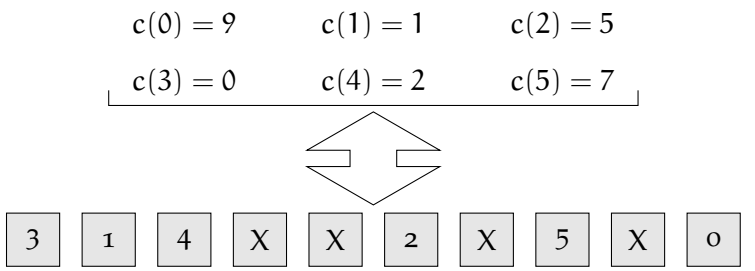

Figure 5.1: Example of representation for an assignment c.

opposite direction of the expected route. A more conventional form of penalty function counts how often the given constraints are broken, and the value of such a π_R is the fraction of successive pairs of acts for all orders that are not sorted with respect to $\precsim$ multiplied by a weight w, which becomes a parameter of the algorithm itself.

Either choice of π_R is used in $e(c)$ by computing their expected value over the dataset $\mathcal{R}$. Unlike t and the first π_R defined above, it is possible to evaluate the second, more conventional penalty function on the entire dataset $\mathcal{R}_F$ as opposed to the sample $\mathcal{R}$, thus eliminating the need for a sample-based estimate for it. However, we chose to compute this expected value over the same sample $\mathcal{R}$ used for t to maintain consistency in the dataset used throughout $e(c)$, as well as to limit computation time; while it is possible to use $\mathcal{R}_F$, it is still very expensive from a computation time standpoint.

5.4 EVOLUTION

As described in Section 5.2.2, GAs are very generic; this section describes the choices made to encode the problem at hand in a way that can be manipulated by standard operators known to the literature.

5.4.1 Representation

To represent a storage assignment $c \in S$, which is an injective function $c : \mathcal{I} \to \mathcal{P}$, we choose to use a list of length $|\mathcal{P}|$, where each position in the list corresponds to a position in $\mathcal{P}$; the element at position $p \in P$ in the list will be SKU $i \in \mathcal{I}$ if $p = c(i)$ for the configuration function and a null value if no such i exists. An example of such a representation is shown in Figure 5.1; the storage assigment c is defined in the upper half of the figure and its representation is the list composed of the elements in the lower half, where the leftmost position in the list corresponds to picking position 0 and the rightmost to position 9. The value in each cell is the SKU $i \in I$ stored in that picking position; X denotes a null element, i. e. a position where no SKU is stored.

For the unconstrained problem this representation is sufficient in that any modification to it that maintains the invariant that each SKU i appears exactly once in the list also corresponds to a valid storage assignment $c' \in S$. For the constrained problem, i. e. when limiting the possible configurations to the subset $S' \subseteq S$, suitable modifications must be made to preserve this property.

For positional constraints, we shall assume that the set of all possible positions for each SKU $D = \{P(i) : i \in \mathcal{I}\}$ forms a finite *partition* of $\mathcal{P}$; that is to say, for any two SKUs $i, j \in \mathcal{I}$, either the two can occupy exactly the same positions ($P(i) = P(j)$) or they must occupy entirely different positions ($P(i) \cap P(j) = \emptyset$). In this case the representation can be amended to be a sequence of lists of length $|D|$, each corresponding to one of the subsets in D and containing all SKUs $i \in \mathcal{I}$ whose position set $P(i)$ is equal to that subset. All operators that are defined for the unconstrained assignment will then act on each list of the sequence independently.

Enforcing hard roller constraints on the representation, on the other hand, requires knowledge of picking order. Due to two-sided aisles like those seen in Section 2.2, this order can only be at most another total preorder $\sqsubseteq$, where $p \sqsubseteq q$ for $p, q \in \mathcal{P}$ if position p is passed by any picker no later than position q according to their picking route. The constraint is then equivalent to enforcing c to be *nonincreasing* with respect to $\precsim$ and $\sqsubseteq$; that is, if $i \precsim j$ then $c(j) \sqsubseteq c(i)$.

To construct a representation that ensures this property it is enough to consider the equivalence relation induced by $\precsim$, namely $i \simeq j \iff i \precsim j \wedge j \precsim i$; by definition $i \simeq j$ if and only if i and j may be picked in any order with regards to one another. Denoting the equivalence class of SKU i according to $\simeq$ as $[i]$, we have that $\precsim$ induces a relation (notated here with the same symbol) over all $[i]$; this relation is not only a total preorder as $\precsim$, but since any two equivalent SKUs are part of the same class, it is also antisymmetric: thus it is a *linear order* on the set $E = \{[i] : i \in \mathcal{I}\}$.

The problem is then reduced to finding a map P' from E to a partition of the position set $\mathcal{P}$ which is similarly linearly ordered with respect to $\sqsubseteq$, and such that $|P'([i])| \geqslant |[i]|$; that is, such that the SKUs in $[i]$ fit inside the associated set of positions. This map P' may then be regarded in the same way as a positional constraint P, and dealt with in the same way — by splitting the representation into a sequence, in this case ordered by $\sqsubseteq$, where each element corresponds to a different $P'([i])$ and contains all SKUs in $[i]$.

5.4.2 Crossover

In order to describe the crossover algorithm we used, we start with a simplifying assumption: that the entire warehouse, or all of its zones as defined in Section 5.4.1, is completely full, i. e. $|\mathcal{P}| = |\mathcal{I}|$. In this case,

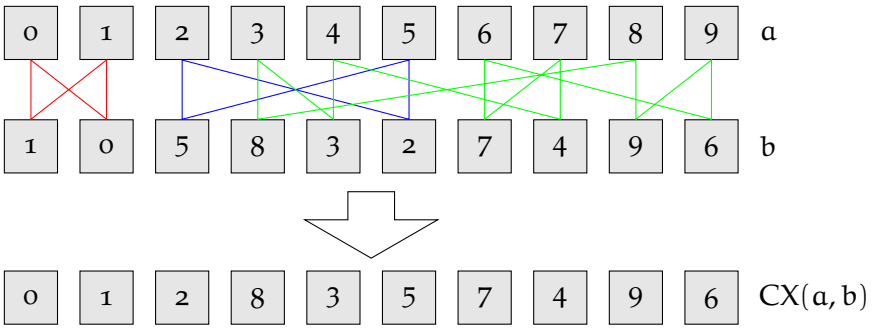

Figure 5.2: Cycle crossover example.

we may regard $\mathcal{P}$ and $\mathcal{I}$ as the same set, since SKU and position labels are entirely arbitrary; any assignment c would then be a *permutation* of that set.

Multiple crossover operators are known for permutations, most of which structure-preserving; they mainly differ in what *aspect* of the permutation is reflected in the offspring: some operators preserve the relative order of SKUs, such as PMX (Goldberg and Lingle, 1985) — that is, the order of SKUs will be a mixture of that of its parents — whereas others, most notably *cycle crossover* (Oliver, Smith and Holland, 1987), preserve the absolute position of SKUs — the mapping c of the offspring will have part of its values from one parent and part from the other. The latter was chosen as the basis for our crossover operator as it can be more readily extended to the case of $|\mathcal{P}| \geqslant |\mathcal{I}|$.

Cycle crossover constructs an offspring by finding partitions of $\mathcal{I}$ (and therefore $\mathcal{P}$) such that for parent permutations a and b and for each such partition $C \subseteq I$ it holds that $a[C] = b[C]$. It is then possible to choose for the offspring permutation $CX(a, b)$ either $[CX(a, b)](i) = a(i)$ or $[CX(a, b)](i) = b(i)$ for all $i \in C$, thus keeping the same positions of all elements of C from one of the parents, while keeping the result a permutation. In particular, the partition to which $i \in \mathcal{I}$ belongs is the *cycle* formed by repeated application of $a^{-1} \circ b$ to it:

$$C_i = \left\{ \left[a^{-1} \circ b\right]^k (i) \,:\, k \in \mathbb{N} \right\};$$

since $\mathcal{I}$ is finite and a and b are permutations it must hold that $\left[a^{-1} \circ b\right]^k (i) = i$ for some $k > 0$, hence the use of the term "cycle." It is evident that all $j \in C_i$ have $C_j = C_i$ by construction, and so $\{C_i \,:\, i \in \mathcal{I}\}$ is indeed a partition of $\mathcal{I}$. Furthermore we have that, since $C_i = \left(a^{-1} \circ b\right)[C_i]$,

$$b\,[C_i] = \left(a \circ a^{-1} \circ b\right)[C_i] = a\,[C_i]\,,$$

which proves that this partition occupies the same positions in both a and b, as required.

Figure 5.2 shows an example of cycle crossover in action. The three cycles defined by a and b are 0–1–0 (red), 2–5–2 (blue) and 3–4–7–6–

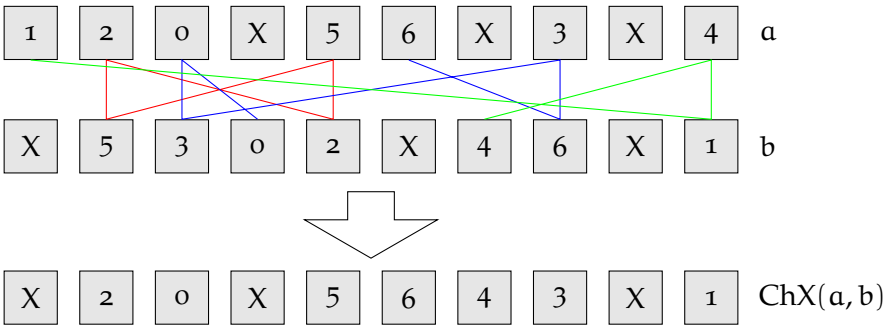

Figure 5.3: Chain crossover example.

9–8–3 (green); the offspring inherits the position of the red and blue cycles from a and of the green cycle from b.

Turning back to the general case, neither parent can be a bijective function if $|\mathcal{P}| > |\mathcal{I}|$; the same process as cycle crossover, if started on any SKU i, may proceed in one of two ways: either as the cycle above, where for some $k > 0$ we have that $\left[a^{-1}b\right]^k(i) = i$, or as a *chain*, terminating as soon as a^{-1} is not defined — that is, when there is no $j \in \mathcal{I}$ such that $a(j) = b(\ell)$ for some ℓ in the chain.

It is nevertheless possible to recover the same partitioning property by starting any such chain on SKUs $i \in \mathcal{I}$ for which no $j \in \mathcal{I}$ exists such that $a(i) = b(j)$; Figure 5.3 shows an example of two such chains (in green and blue, respectively). The green chain starts at SKU 1, which is assigned by a to the first position, to which b assigns no SKU, and ends at SKU 4, for which the reverse applies. For a chain C_i we then have that

$$|a[C_i] \cup b[C_i]| = |C_i| + 1,$$

since the above union comprises the positions of SKUs C_i in both parents plus an unassigned position (at the beginning or end of the chain, respectively). Thus one may freely choose which assignment of SKUs of C_i the offspring should inherit in the same way as cycle crossover.

We call this operator *chain crossover*, or ChX. In the example in Figure 5.3, it is applied so that chain 1–4 is taken from b, whereas chain 6–3–0 and cycle 2–5–2 are from a.

5.4.3 Mutation

The mutation operator is, in principle, meant to act on single genes of the individual's chromosome, letting the ultimate result of such a mutation on the solution it encodes be part of the representation and of its interpretation. For permutations and similar representations, however, this is not possible, as the invariant that each element (in our case, SKU) appear only once is a property of the entire chromosome;

thus any permutation must be considered a single gene and mutated at once.

Due to this limitation a great deal of mutation operators for such representations are available in the literature; these include *swap* operators that select two elements and swap them (whether adjacent or from any position); *scramble* operators that pick a subset of elements and shuffle them; and *reordering* operators that similarly pick a subset of elements and either shift them in one direction or reverse their order. Of these, the most commonly used are swap and scramble operators, as they allow chromosomes to more quickly explore the search space.

Our choice was to use a scramble operator that operates on each partition of the position set independently as per Section 5.4.1, as it is the most similar to traditional operators for GAs, which act on *binary* representations. In their case, mutation strength is controlled by a mutation (bit flip) probability p_m, the effects of which on solution quality have been studied; swap operators must pick pairs to act upon, and therefore the probability that any element of the permutation change due to the operator cannot be controlled directly as a parameter, which is usually the maximum number of pairs to be swapped instead. The latter, having been less of a subject of study, is more often tuned to the requirements of the problem at hand, whereas for the former heuristics are known.

5.4.4 Selection

As explained in Section 5.2.2 it is possible to use the same selection methods for both the parent and replace operators; however, most operators that have been introduced in the literature were designed for only one of the two, and so were subsequently employed. Furthermore, not all characteristics are shared between them — for example, while several several purely deterministic replacement operators exist (such as taking the μ best solutions out of the $\lambda + \mu$ total set of offspring and current generation), deterministic parent selection is extremely uncommon. We thus follow established practice and make use of two different operators for the two tasks.

For parent selection, the original algorithm used by the paper that introduced GA is known as *fitness-proportional selection*; as the name implies, it picks each individual for its output set one at a time, with a probability for each one that is proportional to their fitness. While this operator is straightforward in its implementation, it suffers from multiple issues; the main one, which remains even with modifications to it that solve the others, such as the *sigma-scaling* approach (Goldberg, 1989), is that of *premature convergence*: the operator tends to favor individuals with high fitness too much, leading to successive generations being mostly the offspring of the fittest individual or individuals,

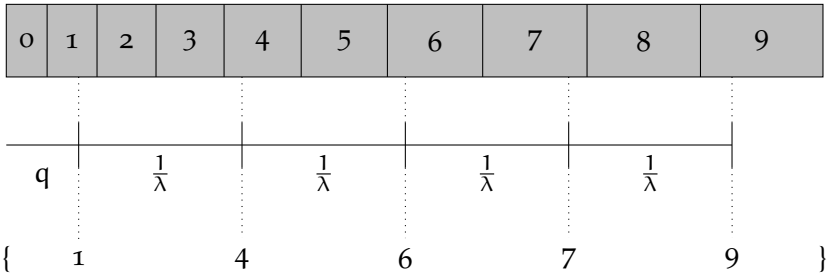

Figure 5.4: Stochastic universal sampling example.

therefore reducing diversity and narrowing the search space towards that local optimum.

For this reason we chose to use a different operator, *stochastic universal sampling*, first described in Baker, 1987. This algorithm is part of a class known as *ranking selection*, as it does not consider the actual value of each individual's fitness, but rather sorts the population according to it and assigns selection probabilities based on their position. To do so a linear function is used, namely

$$p(k) = \frac{(1-s)\cdot(\mu-1)+s\cdot 2k}{\mu(\mu-1)},$$

where k is the individual's position, in the range $[0, \mu-1]$ with 0 being the least-fit. s is a parameter that controls the operator's *selection pressure*; for $s=0$, selection does not consider fitness at all and gives all individuals the same probability of $\frac{1}{\mu}$, wheras for $s=1$ probability is directly proportional to k.

Stochastic universal sampling uses these probabilities in the following way: it first generates a value q uniformly at random in the range $\left[0, \frac{1}{\lambda}\right[$, then picks λ SKUs so that, for SKU $\ell \in [0, \lambda-1]$, the individual at index k is picked if

$$0 \leqslant \left(q + \frac{\ell}{\lambda}\right) - \sum_{j=0}^{k-1} p(j) < p(k);$$

in other words, equally-spaced points at $q + \frac{\ell}{\lambda}$ are mapped to indices according to the inverse of the cumulative probability distribution defined by $p(k)$. Figure 5.4 shows an example of this process with $s = \frac{1}{2}$, $\mu = 10$ and $\lambda = 5$; conceptually, this models rolling a roulette with λ equally-spaced arms, with each of the μ possible results spaced according to p. This selection operator ensures that any individual has a non-zero probability of being chosen, and furthermore that the total number of times the individual at index k appears in the output is no less than $\lfloor \lambda p(k) \rfloor$ and no more than $\lceil \lambda p(k) \rceil$.

As for replacement, i. e. selection of survivors, we employed the *round-robin tournament* strategy: the fitness of each individual in the population is evaluated, then a number of them (the *tournament size*

Module	SKUs	Available	Util. %
Tall	428	762	56.17%
Medium	589	843	69.87%
Short	696	843	82.56%
Roller	285	324	87.96%
Total	1998	2772	72.08%

Table 5.1: New warehouse item database statistics.

	Class												
Module	0	1	2	3	4	5	6	7	8	9	10	11	12
Tall	41	17	4	5	17	0	4	70	37	68	47	59	59
Medium	82	31	14	26	23	8	23	87	26	83	44	51	91
Short	41	45	13	32	38	13	29	44	31	60	76	84	164
Roller	24	22	10	17	11	11	34	23	12	21	29	15	56
Total	214	115	41	80	89	32	90	224	106	232	196	209	370

Table 5.2: New warehouse roller constraint statistics.

z) is picked uniformly at random for each as its *opponents*, and the final population is given by the top μ individuals according to the number of opponents each of them beats in fitness. Tournament size is a parameter of the operator.

Round-robin was chosen for survivor selection because stochastic universal sampling does not provide sufficient selection pressure with regards to offspring fitness. Since the number of times a given individual will be selected as parent is nearly fixed for each generation, it is necessary to provide means for offspring of less fit individuals to survive to the next generation in order to maintain diversity; forms of tournament selection are used in algorithms related to GA such as evolutionary programming as their only source of selection pressure.

5.5 RESULTS

The item database for the new warehouse consists of 1998 SKUs, whose basic statistics according to their module are shown in Table 5.1. Each item can only be allocated picking positions that correspond to its module; this is the main hard positional constraint we faced.

For the same set of SKUs, a total linear order was defined according to the method described in Section 5.4.1; for the constraints given to us this led to a total of 13 equivalence classes, whose statistics are shown

in Table 5.2. Each class is to be placed in the warehouse in numerical order, with 0 being the class closest to the start of the route and 12 at the end.

However, the requirement that each class be assigned separate areas of the warehouse in which all SKUs belonging to it must fit prevented us from enforcing all roller constraints as hard, for in that case later classes (according to the ordering) would not be able to fit inside the warehouse at all. Exhaustive search proved that the finest grouping of classes that could fit meant separating class 0, classes 1 to 6 and classes 7 to 12 in three separate partitions (areas) of the warehouse, enforcing the constraint between their constituent classes as soft.

As written in Section 5.3, the dataset $\mathcal{R}_F$ used for training and validating the predictor is the same used as a base for our optimization process. Since only a small fraction of the acts contained in this dataset (less than 0.1%) featured pickers moving backwards in their picking route, the reordering penalty π_R defined in the same section could not be used with confidence; thus a weighted fraction of act pairs that broke roller constraints was used instead.

One further issue with the original dataset is that the constraints given to us related to the SKU inventory at current time, whereas the order reports dated to several months prior. As a result, not all orders could be used, as some items were only present in one of the two; thus the $\mathcal{R}_F$ used for optimization consisted of a large subset of the dataset used for training the predictor, consisting of 217935 orders for a total of 5584666 acts. Furthermore, since orders were split by zone in the old warehouse so were the orders in $\mathcal{R}_F$, and thus the dataset was not entirely representative of orders as they would be picked in the new warehouse, where no such zoning exists; since no information was available regarding which split orders came from the same customer, there was no way to counteract this.

The shared mutation operator was chosen as scrambling with odds of picking each position in each partition $D \subseteq \mathcal{P}$ equal to $\frac{20}{|D|}$, thus averaging a set of 20 positions per application.

In the following, all average picking times shown were computed over the entire dataset $\mathcal{R}_F$; while it was explained in Section 5.3 that this is not feasible during the optimization process, it is possible to do so for the best configuration found by either algorithm. The same applies for π_R: each order in $\mathcal{R}_F$ is sorted according to the fixed picking route of the new warehouse and the configuration c to be evaluated, and π_R is the fraction of successive pairs of acts belonging to the same order whose picked items are not sorted with respect to the equivalence classes shown above; this fraction is displayed as a percentage.

Both algorithms were given limited computing time, i. e. a fixed number of rounds r for SA and a fixed number of generations g for GA. To gauge their ability to provide good solutions to the problem a

	Configuration		
	Random	Original	Optimized
Pick time	mean 34.603, std. dev. 0.230	32.318	30.011
Constraint breakage	mean 34.771%, std. dev. 0.249%	39.661%	29.306%

Table 5.3: Baseline statistics for the optimization problem.

few baseline statistics were obtained to which their results may be compared; these are shown in Table 5.3 and described below.

The *Random* column was computed by generating 1000 configurations and evaluating the average act picking time t and the unweighted constraint breakage fraction π_R over the whole dataset $\mathcal{R}_F$; the values shown are the mean and standard deviation of these values over the aforementioned set of random configurations. We would like the output of either algorithm to provide better results in at least one of the two, for otherwise the configuration they returned would be no better than one generated at random.

The *Original* column was computed by evaluating the average of the same two functions, but using the position data and the picking order of the old warehouse, as present in the original dataset $\mathcal{R}_F$; we note that this picking time is not equal to that reported by the WMS and used in training the predictor, whose average is given in Section 4.4. These values are provided to compare how well our algorithms fare against the original configuration present in the old warehouse, which was optimized by hand to minimize picking time.

Finally, the *Optimized* column was computed by running a GA with the same fixed parameters as those described below in Section 5.5.2 with $\lambda = 600$, and using as its evaluation function $e(c)$ either only t or only π_R for pick time and constraint breakage, respectively. For t, a number of order samples $\nu = 50000$ was used, whereas π_R was computed over the whole dataset. We consider these "target" values in the sense that, being obtained by only optimizing for one of the two functions, we do not expect algorithms that optimize $e(c)$ for the constrained assignment problem to provide solutions that are as optimized for either.

5.5.1 Simulated annealing

Threshold accepting was run varying a series of parameters:

- the number of orders sampled per evaluation $\nu \in \{100, 1000\}$, with lower values considered too small to estimate the function consistently and larger values taking too long to compute.
- penalty weight $w \in \{10, 100, 1000\}$; since this percentage is about 33% and time is approximately 34 seconds in random configura-

	w		
κ	10	100	1000
$e^{-0.025}$	(34.285, 33.680%)	(34.560, 33.685%)	(34.593, 33.388%)
$e^{-0.05}$	(34.566, 33.846%)	(34.084, 33.480%)	(34.641, 33.296%)
$e^{-0.1}$	(34.184, 33.832%)	(34.217, 33.486%)	(34.931, 33.397%)

Table 5.4: SA results for $\nu = 100$.

	w		
κ	10	100	1000
$e^{-0.025}$	(34.251, 34.175%)	(34.684, 33.566%)	(34.936, 33.213%)
$e^{-0.05}$	(34.023, 33.892%)	(34.470, 33.233%)	(34.734, 33.438%)
$e^{-0.1}$	(34.514, 34.184%)	(34.210, 33.419%)	(34.700, 33.561%)

Table 5.5: SA results for $\nu = 1000$.

tions, a value of 100 would give approximately equal weight to both when optimizing.

- cooling rate κ, with $-\log \kappa \in \{0.1, 0.05, 0.025\}$ — this being an artifact of the temperature being represented as its logarithm when the algorithm was implemented.

Initial temperature was set to $T_0 = e$ for $w \neq 1000$ and $T_0 = 10e$ for $w = 1000$ to allow the solution to escape a local minimum within the scale of the evaluation function.

To prevent the algorithm from being subjected to a different sample of orders $\mathcal{R}$ at each iteration, which would force a change in the optimal trajectory, we set $b = 10$; that is, order samples were drawn only once every 10 iterations. To allow the algorithm to adapt to this new sample, temperature was increased by κ^{-5} (that is, $\sqrt{\kappa^b}$) after drawing $\mathcal{R}$.

The algorithm was run ten times for each parameter set, each time for $r = 1000$ rounds (iterations), and the best solutions found are shown in Table 5.4 and Table 5.5 for a number of samples ν equal to 100 and 1000, respectively. It is evident from these results that SA is not able to cope with the approximation of the evaluation function inherent in sampling the dataset, as its results are all broadly equal to one another, and in particular picking times are essentially the same as those found generating random solutions, as in Table 5.3.

This limitation was already noted by Nissen and Propach, 1998, who compared "point-based" metaheuristics including threshold accepting and "population-based" metaheuristics including GA in the context of a noisy evaluation function and show that the latter are better able to cope with the noise by virtue of maintaining multiple solutions at the

same time. Considering the approximation error as noise, this result is in line with their findings.

It may be also argued that much like threshold accepting performs just as well if not better than standard SA due to randomness being already inherent in mutating the current solution, and thus further randomness in accepting a less-fit solution not being needed, we similarly have that a noisy evaluation function already provides sufficient randomness to allow escape from a local minimum, and further random mutation and acceptance of less-fit solutions — as opposed to e. g. searching the solution's neighbors and accepting the first that is more fit, if any — is also not needed. We surmise that it is this that prevents solutions from being much or any better than randomly-generated ones; threshold and noise together lead to what is effectively a random walk in the search space S'.

5.5.2 Genetic algorithm

Operators for the GA as described above offered a wealth of parameters to tune to the problem at hand; due to time constraints, however, the same amount of variations was provided as those for SA. Other parameters were chosen as follows:

- tournament size for survivor selection was fixed at $q = 10$, in accordance with the recommendation of Eiben and Smith, 2015.
- linear selection pressure was used with $s = \frac{1}{2}$ to provide moderate selection pressure while still allowing less-fit solutions to be selected for reproduction, as this would allow the algorithm to cope with changes in the evaluation function due to different samples.
- population size was fixed at $\mu = 200$; while this is lower than usual for GAs, the much longer time required to evaluate a single generation for our case prevented larger populations from completing their runs in reasonable amounts of time.

Parent selection was implemented by stochastic universal sampling to choose a pool of 2λ parents, with each pair picked uniformly at random without replacement from the generated pool. Additionally, unlike the previous case, we set $b = 1$; thus a different sample of orders $\mathcal{R}$ was picked each generation, and the entire population, as well as its offspring, was evaluated according to it.

w and ν were allowed to vary in the same way as SA and with the same values as before. The number of offspring λ also took on multiple values, replacing the cooling rate, though always as a multiple of μ, and namely $\lambda \in \left\{\frac{1}{4}\mu, \frac{1}{2}\mu, \mu, 2\mu\right\}$.

The algorithm was run once per parameter set, each time for $g = 100$ generations, and the best solutions found in any generated population

$\frac{\lambda}{\mu}$	w = 10	w = 100	w = 1000
$\frac{1}{4}$	(33.600, 32.565%)	(33.810, 32.543%)	(34.334, 32.218%)
$\frac{1}{2}$	(32.952, 32.840%)	(33.353, 32.615%)	(34.212, 32.353%)
1	(32.421, 32.609%)	(33.119, 32.257%)	(33.967, 32.382%)
2	(31.621, 32.853%)	(33.147, 31.414%)	(34.131, 32.108%)

Table 5.6: GA results for $\nu = 100$.

$\frac{\lambda}{\mu}$	w = 10	w = 100	w = 1000
$\frac{1}{4}$	(32.777, 33.218%)	(33.465, 32.473%)	(34.151, 32.406%)
$\frac{1}{2}$	(32.362, 32.826%)	(32.778, 32.249%)	(33.691, 31.521%)
1	(31.885, 32.446%)	(32.811, 31.312%)	(34.111, 31.117%)
2	(31.090, 32.492%)	(32.265, 30.361%)	(33.601, 29.873%)

Table 5.7: GA results for $\nu = 1000$.

for the two sample sizes are shown in Table 5.6 and Table 5.7 for $\nu = 100$ and $\nu = 1000$, respectively.

In contrast to SA, our GA implementation provides values for both functions to be optimized that are noticeably better than random solutions; this is evident when optimizing with either low w — i. e. weighing picking time more strongly over constraints — or with $\nu = 1000$: the case $\lambda = 1, w = 10, \nu = 1000$, for example, is about 12 standard deviations lower than the mean of random configurations as per Table 5.3 when it comes to picking time and over 9 lower for constraint breakage. Similar results hold for many of the other cases, and especially for high values of λ — though it must be noted that, as we shall further explore below, for $\nu = 100$ the algorithm struggles to obtain good solutions when $w \geqslant 100$. Nevertheless, these results generally track the intended goal of the optimization as given by w; larger weight values result in configurations that minimize constraint breakage at the expense of picking time and vice versa.

As already cited, this contrast was already noted in a different optimization task by Nissen and Propach, 1998; it also appears to be the case here that, keeping with the biological metaphor of evolution, population-based algorithms such as our GA implementation are better able to adapt to a "varying environment:" this being, in our case, the order sampling process, which leads to different optimization paths to be taken through the search space S'. The output population at the end is thus more "robust" than the single output of SA, as its

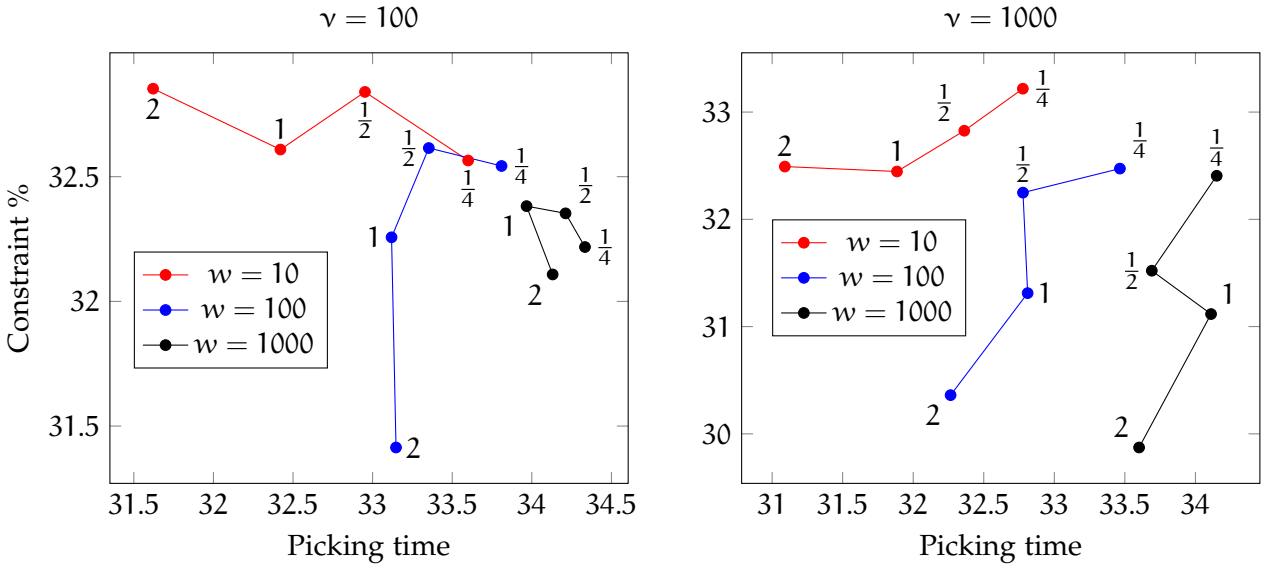

Figure 5.5: Trajectories as a function of $\frac{\lambda}{\mu}$.

individuals have developed the adaptations needed to survive these abrupt changes.

Looking back at the data, from the values above we note that while higher values of ν make for a more stable optimization process, it is higher values of λ that provide a clear advantage in solution quality. This is especially the case for $w = 1000$, where there is scarcely any difference in constraint percentage between $\lambda = \frac{1}{4}\mu$ and $\lambda = 2\mu$ when $\nu = 100$, whereas for the latter value and $\nu = 1000$ the result obtained is close to what one would obtain optimizing only for that value, as shown earlier.

This is even more clearly seen by plotting the trajectories taken by the algorithm when optimizing with different values of λ; this was done in Figure 5.5, with the values of $\frac{\lambda}{\mu}$ annotated near each point and the points connected by increasing λ. For $\nu = 100$, the trajectory for $w = 10$ is as expected, but the other two weights either optimize for only one metric ($w = 100$) or do not seem to go anywhere ($w = 1000$). A larger sample value of $\nu = 1000$ leads to all paths looking as expected: $w = 10$ goes in the direction of lower picking time, only lowering constraint breakage somewhat; $w = 100$ is able to decrease both more or less at the same rate; and even $w = 1000$, while still faring not much better than $w = 100$ for constraints, is still able to optimize for the latter.

Another concern is whether or not after 100 iterations the resulting population has encountered *genetic drift*; this occurs when a local minimum is found and, unable to find other similar solutions, the rest of the population is slowly filled with copies or near-copies of the same individual that encodes for the minimum. This loss of diversity leads to diminishing or no returns when increasing the number of

generations, and is generally counteracted by either a larger number of offspring λ or by additional rules regarding parent selection that prevent similar individuals from mating.

To show that GA as implemented here is not subject to this issue an additional two runs were made with $\lambda = 3\mu$ and $w = 30$, with ν values as before, and two-dimensional histograms of the final population with respect to the two metrics were plotted in Figure 5.6 and Figure 5.7. The distributions of both, as can be seen, are diverse; this is the more the case for $\nu = 100$, where the even noisier fitness computation naturally leads to populations with greater variance, whereas the result for $\nu = 1000$ is more concentrated. We conclude that the sampling process alleviates this common GA pitfall and that even for $g = 200$ there does not seem to be a convergence towards a single optimal "strain" of solutions, but rather a larger set of candidates in the same population.

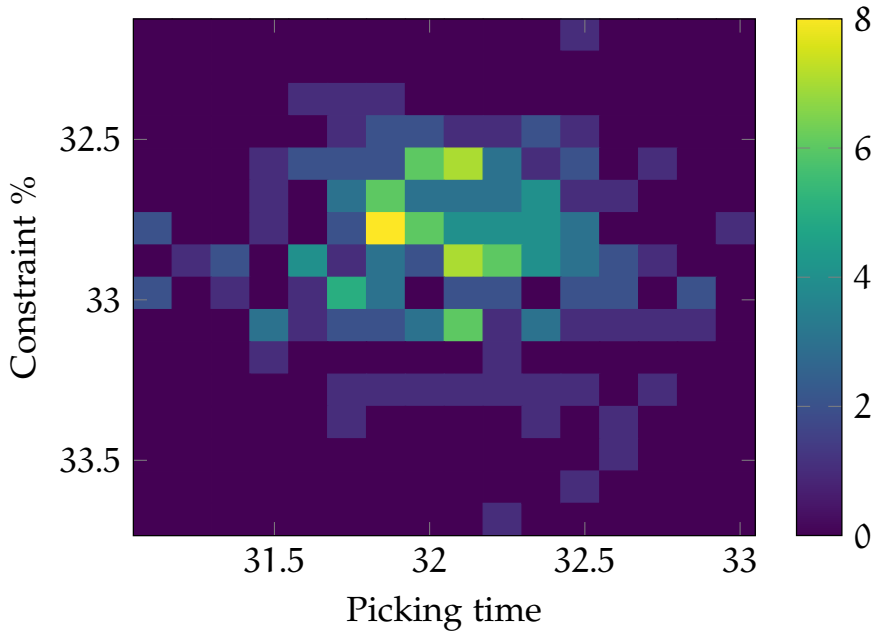

Figure 5.6: Two-dimensional histogram for $\nu = 100$.

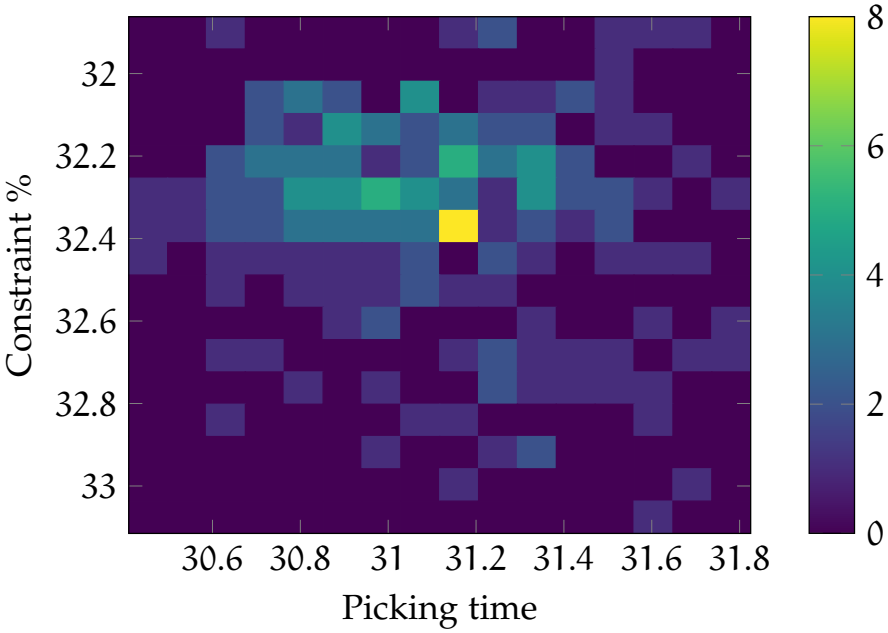

Figure 5.7: Two-dimensional histogram for $\nu = 1000$.

6 CONCLUSIONS

In this thesis we have investigated the ability of a custom ML architecture in the form of a RNN to deal with warehouse pick time prediction in service of its use as a component of an optimization algorithm for assigning locations to the products stored therein.

Our work began with an analysis of warehouse order data as obtained by a real warehouse; this data was then processed for use with ML tools to predict picking times. The choice of what to encode and how was crucially dependent on the task at hand, and most importantly on the fact that this data referred to a different warehouse than the one we would then have to optimize storage for. This arrangement is rarely found in either literature or practice, as it is more common to obtain data for the new warehouse and use that to optimize storage.

One further issue concerns the relatively atypical format of zones in the old warehouse, which not only limited the dataset to single-zone orders, but also prevented the prediction model from learning how pickers move in one united warehouse, as opposed to multiple zones with separate paths and orientations.

The problem of pick time prediction was then specified formally, and a RNN-based prediction model was developed to tackle it. The latter fared well when faced with pickers moving around the same area of the warehouse, but its accuracy was not up to par when moving between areas, due to the reasons listed above.

Finally, the storage location assignment problem was described and two algorithms were presented for its approximate solution; one SA variant and GA. Despite the limitations of the predictor and of the available data, the GA-based optimization model was still able to provide solutions whose performance is superior to the old warehouse and is close to optimal with respect to the given constraints and the limited computation time available.

The results thus obtained are promising, as we were able to show that algorithms such as GA are well-suited for optimization involving approximate computation of picking time and constraints, both in the sense of predicting the former and in estimating both by sampling.

6.1 FUTURE RESEARCH

As mentioned above, one key issue was the disconnect between prediction data and optimization goals; it would be beneficial to apply a

similar process to the one described here to optimize allocation for the same warehouse. In that case, position information in the form of X–Y coordinates would not be needed, as the structure of the warehouse would be static during the whole process; a simpler one-hot encoding of aisles and bays may work as well if not better.

Other features may also be extracted from picker data, such as those outlined in Section 3.3.1; exerted work, in particular, is particularly attractive, and is easily derived and used for a warehouse with static structure. More generally, integrating more data for both orders and items may be beneficial.

Much experimentation is also possible within the prediction model itself; one may e. g. predict picking *speed* directly and not time, with a different output distribution, and then generate a time distribution from it knowing the distance that the picker must travel. In keeping with the model as described here it would also be interesting to explore the hyperparameter space further, though as already mentioned our specific application should not be used as a reference model as it is an uncommon scenario.

As for optimization, like the prediction model, the general framework allows for much variation. Other population-based algorithms may be used, as well as different operators or parameters for the same. Similarly, in keeping with the observations regarding SA one may apply a mutation step that directly performs its own optimization — such as SA — and thus assume that genetic drift may already be prevented efficiently by the sampling process.

BIBLIOGRAPHY

Ansari, Mohammadnaser and Jeffrey S. Smith (2020). 'Gravity Clustering: A Correlated Storage Location Assignment Problem Approach'. In: *2020 Winter Simulation Conference (WSC)*, pp. 1288–1299. DOI: 10.1109/WSC48552.2020.9384029.

Atmaca, Ediz and Ayla Ozturk (2013). 'Defining order picking policy: A storage assignment model and a simulated annealing solution in AS/RS systems'. In: *Applied Mathematical Modelling* 37.7, pp. 5069–5079. ISSN: 0307-904X. DOI: 10.1016/j.apm.2012.09.057.

Baker, J. E. (1987). 'Reducing bias and inefficiency in the selection algorithm'. In: *Proceedings of the 2nd International Conference on Genetic Algorithms and Their Application*. Cambridge, Massachusetts: Lawrence Erlbaum, pp. 14–21. ISBN: 0805801588.

Balzereit, Kaja, Nehal Soni and Andreas Bunte (2023). 'Potentials of Explainable Predictions of Order Picking Times in Industrial Production'. In: *Proceedings of the 15th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART,* INSTICC. SciTePress, pp. 405–412. ISBN: 978-989-758-623-1. DOI: 10.5220/0011677000003393.

Bengio, Y., P. Simard and P. Frasconi (1994). 'Learning long-term dependencies with gradient descent is difficult'. In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166. DOI: 10.1109/72.279181.

Bottani, Eleonora et al. (2012). 'Optimisation of storage allocation in order picking operations through a genetic algorithm'. In: *International Journal of Logistics Research and Applications* 15.2, pp. 127–146. DOI: 10.1080/13675567.2012.694860.

Cho, Kyunghyun et al. (Oct. 2014). 'Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, pp. 1724–1734. DOI: 10.3115/v1/D14-1179.

Chung, Junyoung et al. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv: 1412.3555 [cs.NE]. URL: https://arxiv.org/abs/1412.3555.

Coyle, J.J., E.J. Bardi and C.J. Langley (1996). *The Management of Business Logistics*. West Publishing Company. ISBN: 9780314065070.

Daria Battini Martina Calzavara, Alessandro Persona and Fabio Sgarbossa (2015). 'Order picking system design: the storage assignment and travel distance estimation (SA&TDE) joint method'. In: *International Journal of Production Research* 53.4, pp. 1077–1093. DOI: 10.1080/00207543.2014.944282.

Dueck, Gunter and Tobias Scheuer (1990). 'Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing'. In: *Journal of Computational Physics* 90.1, pp. 161–175. ISSN: 0021-9991. DOI: 10.1016/0021-9991(90)90201-B.

Eiben, A. E. and J. E. Smith (2015). 'Fitness, Selection, and Population Management'. In: *Introduction to Evolutionary Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 79–98. ISBN: 978-3-662-44874-8. DOI: 10.1007/978-3-662-44874-8_5.

Fontana, Marcele Elisa et al. (2020). 'Multi-criteria assignment model to solve the storage location assignment problem'. In: *Revista de Investigación Operacional* 41.7, pp. 1019–1029. ISSN: 2224-5405.

Franz, Astrid, Karl Heinz Hoffmann and Peter Salamon (2001). 'Best Possible Strategy for Finding Ground States'. In: *Phys. Rev. Lett.* 86 (23), pp. 5219–5222. DOI: 10.1103/PhysRevLett.86.5219.

Gino Marchet, Marco Melacini and Sara Perotti (2015). 'Investigating order picking system adoption: a case-study-based approach'. In: *International Journal of Logistics Research and Applications* 18.1, pp. 82–98. DOI: 10.1080/13675567.2014.945400.

Glorot, Xavier and Yoshua Bengio (2010). 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

Goldberg, David E. and Robert Lingle (1985). 'Alleles, loci, and the traveling salesman problem'. In: *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*. Ed. by J. J. Grefenstette. Hillsdale, New Jersey, USA: Lawrence Erlbaum, pp. 154–159.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley series in artificial intelligence. Addison-Wesley. ISBN: 9780201157673.

Guerriero, F. et al. (2013). 'A mathematical model for the Multi-Levels Product Allocation Problem in a warehouse with compatibility constraints'. In: *Applied Mathematical Modelling* 37.6, pp. 4385–4398. ISSN: 0307-904X. DOI: 10.1016/j.apm.2012.09.015.

Hinton, Geoffrey (2012). *rmsprop: Divide the gradient by a running average of its recent magnitude*. Lecture notes. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). 'Long Short-Term Memory'. In: *Neural Comput.* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

Holland, John H. (Apr. 1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press. ISBN: 9780262275552. DOI: 10.7551/mitpress/1090.001.0001.

Hornik, Kurt, Maxwell Stinchcombe and Halbert White (1989). 'Multilayer feedforward networks are universal approximators'. In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

Islam, Md. Saiful and Md. Kutub Uddin (2023). 'Correlated storage assignment approach in warehouses: A systematic literature review'. In: *Journal of Industrial Engineering and Management* 16.2, pp. 294–318. ISSN: 2013-0953. DOI: 10.3926/jiem.4850.

Jiao, Yu ling et al. (2018). 'Multi-objective storage location allocation optimization and simulation analysis of automated warehouse based on multi-population genetic algorithm'. In: *Concurrent Engineering* 26.4, pp. 367–377. DOI: 10.1177/1063293X18796365.

Kabak, Özgür and Nurullah Güleç (2022). 'Data Driven Approach to Order Picking Time Prediction Using Fuzzy Clustering and ANN'. In: *Intelligent and Fuzzy Techniques for Emerging Conditions and Digital Transformation*. Ed. by Cengiz Kahraman et al. Cham: Springer International Publishing, pp. 18–26. ISBN: 978-3-030-85626-7.

Kingma, Diederik P. and Jimmy Ba (2015). 'Adam: A Method for Stochastic Optimization'. In: *Conference Track Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. Ed. by Yoshua Bengio and Yann LeCun. San Diego, CA, USA. URL: http://arxiv.org/abs/1412.6980.

Kirkpatrick, S., C. D. Gelatt and M. P. Vecchi (1983). 'Optimization by Simulated Annealing'. In: *Science* 220.4598, pp. 671–680. DOI: 10.1126/science.220.4598.671.

Le-Duc, Tho and René M.B.M. de Koster (2007). 'Travel time estimation and order batching in a 2-block warehouse'. In: *European Journal of Operational Research* 176.1, pp. 374–388. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2005.03.052.

Lee, In Gyu, Sung Hoon Chung and Sang Won Yoon (2020). 'Two-stage storage assignment to minimize travel time and congestion for warehouse order picking operations'. In: *Computers & Industrial Engineering* 139, p. 106129. ISSN: 0360-8352. DOI: 10.1016/j.cie.2019.106129.

Li, Jiaxi, Mohsen Moghaddam and Shimon Y. Nof (June 2016). 'Dynamic storage assignment with product affinity and ABC classification — a case study'. In: *The International Journal of Advanced Manufacturing Technology* 84.9, pp. 2179–2194. ISSN: 1433-3015. DOI: 10.1007/s00170-015-7806-7.

Mirzaei, Masoud, Nima Zaerpour and René de Koster (2021). 'The impact of integrated cluster-based storage allocation on parts-to-picker warehouse performance'. In: *Transportation Research Part E: Logistics and Transportation Review* 146, p. 102207. ISSN: 1366–5545. DOI: 10.1016/j.tre.2020.102207.

Muppani (Muppant), Venkata Reddy and Gajendra Kumar Adil (2008). 'Efficient formation of storage classes for warehouse storage location

assignment: A simulated annealing approach'. In: *Omega* 36.4. Special Issue on Logistics: New Perspectives and Challenges, pp. 609–618. ISSN: 0305-0483. DOI: 10.1016/j.omega.2007.01.006.

Nissen, V. and J. Propach (Sept. 1998). 'On the robustness of population-based versus point-based optimization in the presence of noise'. In: *Trans. Evol. Comp* 2.3, 107–119. ISSN: 1089-778X. DOI: 10.1109/4235.735433.

Oliver, I. M., D. J. Smith and J. R. C. Holland (1987). 'A study of permutation crossover operators on the traveling salesman problem'. In: *Proceedings of the 2nd International Conference on Genetic Algorithms and Their Application*. Cambridge, Massachusetts: Lawrence Erlbaum, pp. 224–230. ISBN: 0805801588.

Pan, Jason Chao-Hsien, Ming-Hung Wu and Wen-Liang Chang (2014). 'A travel time estimation model for a high-level picker-to-part system with class-based storage policies'. In: *European Journal of Operational Research* 237.3, pp. 1054–1066. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2014.02.037.

Riccardo Accorsi, Riccardo Manzini and Marco Bortolini (2012). 'A hierarchical procedure for storage allocation and assignment within an order-picking system. A case study'. In: *International Journal of Logistics Research and Applications* 15.6, pp. 351–364. DOI: 10.1080/13675567.2012.742877.

Roodbergen, Kees Jan and René de Koster (2001). 'Routing order pickers in a warehouse with a middle aisle'. In: *European Journal of Operational Research* 133.1, pp. 32–43. ISSN: 0377-2217. DOI: 10.1016/S0377-2217(00)00177-6.

Saxe, Andrew M., James L. McClelland and Surya Ganguli (2014). 'Exact solutions to the nonlinear dynamics of learning in deep linear neural networks'. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1312.6120.

Tompkins, J.A. et al. (2010). *Facilities Planning*. Wiley. ISBN: 9780470444047.

van Gils, Teun et al. (2019). 'Designing efficient order picking systems: The effect of real-life features on the relationship among planning problems'. In: *Transportation Research Part E: Logistics and Transportation Review* 125, pp. 47–73. ISSN: 1366-5545. DOI: 10.1016/j.tre.2019.02.010.

Vapnik, V. (1991). 'Principles of risk minimization for learning theory'. In: *Proceedings of the 5th International Conference on Neural Information Processing Systems*. NIPS'91. Denver, Colorado: Morgan Kaufmann Publishers Inc., pp. 831–838. ISBN: 1558602224.

Wang, Ye, Jun Zheng and Su Wang (2011). 'Evolutionary Algorithm Inspired Particle Swarm Optimization (EA-PSO) for Warehouse Allocation Problem'. In: *2011 3rd International Conference on Computer*

Research and Development. Vol. 2, pp. 38–41. DOI: 10.1109/ICCRD.2011.5764079.

Wang, Zheng, Kun Fu and Jieping Ye (2018). 'Learning to Estimate the Travel Time'. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '18. London, United Kingdom: Association for Computing Machinery, pp. 858–866. ISBN: 9781450355520. DOI: 10.1145/3219819.3219900.

Wei Jiang Jiyin Liu, Yun Dong and Li Wang (2021). 'Assignment of duplicate storage locations in distribution centres to minimise walking distance in order picking'. In: *International Journal of Production Research* 59.15, pp. 4457–4471. DOI: 10.1080/00207543.2020.1766714.

Werbos, P.J. (1990). 'Backpropagation through time: what it does and how to do it'. In: *Proceedings of the IEEE* 78.10, pp. 1550–1560. DOI: 10.1109/5.58337.

Yuan, Ruiping, Huiling Wang and Juntao Li (2019). 'The Pod Assignment Model and Algorithm in Robotic Mobile Fulfillment Systems'. In: *2019 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, pp. 99–103. DOI: 10.1109/SOLI48380.2019.8955103.

Yuan, Ruiping et al. (2021). 'Storage Assignment Optimization in Robotic Mobile Fulfillment Systems'. In: *Complexity* 2021.1, p. 4679739. DOI: 10.1155/2021/4679739.

Zhang, Ren-Qian, Meng Wang and Xing Pan (2019). 'New model of the storage location assignment problem considering demand correlation pattern'. In: *Computers & Industrial Engineering* 129, pp. 210–219. ISSN: 0360-8352. DOI: 10.1016/j.cie.2019.01.027.

Zhang, Y. (2016). 'Correlated Storage Assignment Strategy to reduce Travel Distance in Order Picking'. In: *IFAC-PapersOnLine* 49.2. 7th IFAC Conference on Management and Control of Production and Logistics MCPL 2016, pp. 30–35. ISSN: 2405–8963. DOI: 10.1016/j.ifacol.2016.03.006.